TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Diplomarbeit

zur Erlangung des akademischen Grades
Diplomingenieur für Informationssystemtechnik

SGI RASC: Evaluierung einer Programmierplattform zum Einsatz von FPGAs als Hardware-Beschleuniger im Hochleistungsrechnen

Robert Dietrich
(Geboren am 10. August 1983 in Dresden)

Betreuer: Guido Juckeland, Thomas B. Preußer

Dresden, 30. Juni 2009

Hier Aufgabenstellung einfügen!

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Elektrotechnik und Informationstechnik eingereichte Diplomarbeit zum Thema:

SGI RASC: Evaluierung einer Programmierplattform zum Einsatz von FPGAs als Hardware-Beschleuniger im Hochleistungsrechnen

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 30. Juni 2009

Robert Dietrich

Kurzfassung

Im Hochleistungsrechnen wird heutzutage, neben massiver Parallelverarbeitung und entsprechend effizienter paralleler Programmierung, verstärkt spezielle Hardware zur Beschleunigung von Anwendungen verwendet. Üblicherweise wird dabei ein Problem, eine Anwendung oder ein Algorithmus als eine Abfolge von Befehlen (Software) an die Hardware, wie z.B. Mikroprozessoren oder Grafikkarten angepasst. Field Programmable Gate Arrays (FPGAs) bieten mit der Beschreibung von Hardware einen anderen Ansatz, welcher abhängig von der Problemstellung eine effizientere Verarbeitung ermöglicht. Gemessen an der Performance können Hardware-Beschleuniger zudem eine deutlich geringere Leistungsaufnahme als CPUs erreichen, wodurch auch die Betriebskosten sinken. FPGAs erlauben durch die hohe Anzahl von frei programmierbaren Logikblöcken eine sehr feingranulare Parallelverarbeitung und bieten im Vergleich zu ASICs den zusätzlichen Vorteil, die Hardware an verschiedene Probleme anpassen zu können. Zur Programmierung von FPGAs gibt es verschiedene Ansätze: Hardwarebeschreibungssprachen (VHDL, Verilog), Hochsprachen (z.B. Handel-C, Impulse C, Mitrion-C) und graphische Tools (z.B. DSPLogic RC Toolbox). Die Qualität der Lösung ist mitunter sehr verschieden und soll in dieser Arbeit anhand von Benchmarks und Beispielanwendungen in Mitrion-C und VHDL verglichen werden. SGI RASC dient dabei als Programmierplattform und wird hinsichtlich ihrer Leistungsfähigkeit evaluiert.

Abstract

Todays High Performance Computing uses, next to massively concurrent processing and respectively parallel programming, additional hardware to speed up applications. Hence, an algorithm is adapted to hardware, e.g. microprocessors or graphics processing units (GPUs), as a sequence of instructions (software). Field Programmable Gate Arrays (FPGAs) offer a totally different approach through their flexible hardware description, which in cases can be more efficient and increase operation speed. Depending on their performance hardware accelerators can noticeably reduce the overall power consumption. FPGAs provide fine-grained parallelism with a large number of programmable logic blocks and can be adapted to a new task in a fraction of a second. There are multiple approaches to program FPGAs: hardware description languages (Verilog, VHDL), high level languages (e.g. Handel-C, Impulse C, Mitrion-C) and graphic tools (e.g. DSPLogic RC Toolbox). The quality of the resulting hardware designs can in fact be very different, which will be compared in this paper using sample implementations and benchmarks in Mitrion-C and VHDL. Thus, the performance of the programming platform SGI RASC will be evaluated in this context.

Inhaltsverzeichnis

1 Einleitung **5**

2 FPGAs **7**
2.1 Typische Einsatzgebiete . . . 7
2.2 Aufbau und Grundstruktur / Architektur . . . 9
2.3 Stärken und Schwächen von FPGAs . . . 11
2.4 Virtex-4 FPGAs . . . 13
2.5 Fließkommaverarbeitungsleistung . . . 15
2.6 FPGAs als Hardware-Beschleuniger im HPC . . . 19

3 SGI RASC **21**
3.1 SGI Altix 4700 . . . 21
3.2 SGI RC100 FPGA-Blade . . . 23
3.3 Core Services . . . 23
3.3.1 SRAM-Schnittstelle . . . 26
3.3.2 Streaming Engines . . . 27
3.3.3 Memory Mapped Registers . . . 28
3.4 Software . . . 28
3.4.1 Abstraction Layer (API) . . . 29
3.4.2 Algorithmen-Verwaltung . . . 30
3.4.3 GNU Project Debugger (GDB) . . . 31
3.4.4 Algorithmus-Konfigurations-Tool . . . 32

4 Programmiermöglichkeiten der SGI RASC Plattform **33**
4.1 Mitrion SDK von Mitrionics . . . 34
4.1.1 Programmiersprache Mitrion-C . . . 36
4.1.2 Mitrion Virtual Processor . . . 39
4.1.3 Mitrion Simulator . . . 40
4.2 Hardware-Entwicklungsablauf . . . 43
4.2.1 Hardware-Entwurf mit VHDL . . . 45
4.2.2 Simulation . . . 48
4.2.3 Hardware-Synthese . . . 49

5 Fallbeispiele **53**
5.1 SRAM-Schnittstelle . . . 55
5.1.1 Implementierung . . . 55
5.1.2 Tests und Messungen . . . 57
5.2 Direct Streaming . . . 58
5.2.1 Implementierung . . . 58
5.2.2 Tests und Messungen . . . 59
5.3 Kommunikation über Memory Mapped Registers . . . 61

5.3.1 Implementierung . . . 61
5.3.2 Tests und Messungen . . . 62
5.4 MD5 Brute Force . . . 63
5.4.1 VHDL-Entwurf . . . 65
5.4.2 Implementierung mit Mitrion-C . . . 68
5.4.3 Vergleich der MD5-Implementierungen . . . 70
5.5 Damenproblem . . . 71
5.5.1 Backtracking-Algorithmus und Parallelisierung . . . 73
5.5.2 VHDL-Implementierung . . . 74
5.5.3 Implementierung mit Mitrion-C . . . 77
5.5.4 Performance . . . 79

6 Auswertung **81**
6.1 Anwendungsbeschleunigung durch FPGAs . . . 81
6.2 Vergleich von Mitrion-C und VHDL (RASC-Programmierung) . . . 82
6.2.1 Entwicklungszeiten . . . 83
6.2.2 Ease-of-Use . . . 85
6.2.3 Fehleranfälligkeit und Debugging-Möglichkeiten . . . 88
6.2.4 Performance . . . 89
6.3 Kosten-Nutzen-Abschätzung und Fazit . . . 91
6.4 Verbesserungsansätze der RASC-Plattform . . . 93

7 Zusammenfassung und Ausblick **95**

Abbildungsverzeichnis **99**

Tabellenverzeichnis **101**

Auflistungsverzeichnis **103**

Literaturverzeichnis **105**

A Quellcodes **107**
A.1 SRAM-Schnittstelle . . . 107
A.2 Direct Streaming . . . 112
A.3 Memory Mapped Registers . . . 114
A.4 MD5-Brute-Force . . . 118
A.5 Damenproblem . . . 123

Abkürzungsverzeichnis

ADR	Algorithm Defined Register
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Products
BLAS	Basic Linear Algebra Subprograms
CLB	Configurable Logic Block
COP	Koprozessor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DCM	Digital Clock Manager
DGEMM	Double Precision General Matrix Multiply
DLL	Delay-Locked Loop
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access
DSP	Digitaler Signalprozessor
EEPROM	Electrically Erasable Programmable Read Only Memory
FF	Flipflop
FIFO	First In – First Out
FLOP/s	Fließkommaoperationen pro Sekunde
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GDB	GNU Debugger
GEMM	General Matrix Multiply
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
GPRS	General Packet Radio Service
GUI	Graphical User Interface (dt. grafische Benutzeroberfläche)
HDL	Hardware Description Language
HPC	High Performance Computing
I/O	Input/Output (dt. E/A – Eingabe/Ausgabe)
IP	Intellectual Property
LUT	Lookup-Tabelle
MAC	Media Access Control
MD5	Message-Digest Algorithm 5
MMR	Memory Mapped Register
MVP	Mitrion Virtual Processor
NAS	Network Attached Storage
PCI	Peripheral Component Interconnect
PLD	Programmable Logic Device
RAM	Random-Access Memory (dt. Speicher mit wahlfreiem Zugriff)
RASC	Rekonfigurierbares Anwendungs-Spezifisches Computing
SAN	Storage-Area-Network

SDK	Software Development Kit
SGEMM	Single Precision General Matrix Multiply
SoC	System on a Chip
SRAM	Statischer RAM
SSL	Secure Sockets Layer
TCL	Tool Command Language
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL)
ZIH	Zentrum für Informationsdienste und Hochleistungsrechnen

1 Einleitung

Das Hochleistungsrechnen (engl. high performance computing – HPC) hat die Grenze der sequentiellen Verarbeitung bereits vor Jahren erfahren und basiert heute auf massiver Parallelverarbeitung. Dabei ist die akkumulierte Rechenleistung auf eine Vielzahl von Prozessorkernen verteilt und kann nur durch effiziente Parallelisierung von Programmen genutzt werden. Um Ausführungzeiten von Anwendungen zu verkürzen oder wissenschaftliche Berechnungen überhaupt zu ermöglichen, muss demnach ein hoher Aufwand in die effiziente parallele Programmierung investiert werden.

Eine weitere Herausforderung im HPC betrifft den immensen Stromverbrauch aktueller Systeme im Teraflop-Bereich. Wie zukünftige Exaflop-Systeme damit umgehen, wird derzeit unter dem Schlagwort „Green IT“ diskutiert. Allerdings ist es im Moment noch eine offene Frage, wie die erwarteten Leistungssteigerungen ermöglicht werden sollen, ohne den CO_2-Ausstoß zu erhöhen.

Einen Beitrag zur Beantwortung dieser Frage könnten Hardware-Beschleuniger, wie Grafikchips (GPUs) und Field Programmable Gate Arrays (FPGAs) sein. Gemessen an der Performance können beide eine deutlich geringere Leistungsaufnahme als CPUs erreichen. Dennoch sind solche Beschleuniger nicht darauf ausgelegt typische Aufgaben eines Mikroprozessors zu erfüllen (z.B. Ausführung des Betriebssystems, zwingend sequentielle Aufgaben wie Verbindungsaufbau/-abbau oder Lese-/Schreiboperationen auf Festplatten). Zukünftige HPC-Systeme können demnach aus einer Kombination von CPUs und spezialisierten Beschleuniger-Chips bestehen. Aufgabe der HPC-Programmierer bleibt dann die effiziente Verteilung der Arbeitslast vorzunehmen. Dabei könnten sequentielle oder stark begrenzt parallele Aufgaben weiterhin auf Mikroprozessoren gelöst werden, während berechnungsintensive hochparallele Arbeit auf Hardware-Beschleuniger ausgelagert wird.

Im Vergleich zur Parallelverarbeitung auf mehreren Mikroprozessoren, ermöglichen FPGAs eine sehr feingranulare Parallelisierung. Während typische Anwendungsbeschleunigung eine Anpassung an die Hardware (z.B. an Befehlssatzerweiterungen) erfordert, wird bei FPGAs die Hardware an das Problem angepasst und kann binnen Millisekunden umkonfiguriert werden. Heutige FPGAs besitzen genug Logikelemente um eine Vielzahl von 64-Bit Verarbeitungseinheiten (z.B. Addierer, Multiplizierer, etc.) zu integrieren, wobei es dem Programmierer obliegt, andere Bitbreiten und spezialisierte problemspezifische Operationen zu verwenden.

Aktuelle HPC-Systeme arbeiten mit einer Vielzahl von Mikroprozessoren, welche durch ein Netzwerk mit hoher Bandbreite und geringer Latenz miteinander verbunden sind. Während FPGAs in eingebetteten Systemen u.a. Verwendung finden, um mehrere Bauteile durch ein einziges zu ersetzen, werden im HPC mehrere FPGAs auf einer Platine plaziert und an ein Mikroprozessor-basiertes System angeschlossen. Die SGI RASC-Technologie für „**R**ekonfigurierbares **A**nwendungs-**S**pezifisches **C**omputing“ integriert mit dem RC100-Blade FPGAs als Hardware-Beschleuniger in HPC-Systeme der Altix 450/4700 Serie. Welche Anwendungen sich beschleunigen lassen und ob andere Beschleuniger wie Grafikkarten bessere Performance bieten, ist Teil der Untersuchungen dieser Arbeit.

Zur Programmierung von FPGAs gibt es verschiedene Möglichkeiten, welche zu mehr oder weniger effizienten Schaltungen führen. Auf der niedrigsten Ebene kann eine Schaltung mit Hardwarebeschreibungssprachen (HDLs), wie VHDL oder Verilog, beschrieben werden. Diese haben den größten Einfluss auf das Design und bieten für erfahrene Programmierer die beste Performance. Typischerweise sind jedoch HPC-Programmierer meist keine Hardware-Entwickler. Um die FPGA-Programmierung dem HPC-Programmierer dennoch zugänglich zu machen, abstrahieren Hochsprachen von der Komplexität des Hardware-Entwurfs. Durch diese Sprachen kann auf Kosten der Performance eine deutliche Steigerung der Produktivität erreicht werden.

Diese Arbeit soll mit SGI RASC eine HPC-Programmierplattform zum Einsatz von FPGAs als Hardware-Beschleuniger untersuchen und dabei die Hochsprache Mitrion-C im Vergleich zu VHDL evaluieren. Dazu wird grundlegendes Wissen über FPGAs als Zieltechnologie in Kapitel 2 vermittelt, wobei speziell auch die hier verwendeten Virtex-4 FPGAs betrachtet werden. Die Fließkommaverarbeitung ist keine Stärke von FPGAs, soll aber dennoch in Abschnitt 2.5 als typisches Leistungskriterium im HPC untersucht werden. In Kaptitel 3 wird schließlich die SGI RASC Programmierplattform vorgestellt, wobei neben Aufbau und Kenndaten auch auf die Hardware- und Software-Schnittstelle eingegangen wird. Eine grundlegende Einführung in die Programmiermöglichkeiten Mitrion-C und VHDL wird in Kapitel 4 vorgenommen. Die Fallbeispiele, welche die Grundlage der Evaluierung von Mitrion-C bilden, werden in Kapitel 5 beschrieben. Dabei wird die mit VHDL und Mitrion-C erreichte Performance auch mit anderen Technologien verglichen. Die Auswertung in Kapitel 6 soll zum einen das Leistungsvermögen der SGI RASC Plattform bewerten und zum anderen Mitrion-C mit VHDL vergleichen und verdeutlichen, in welchen Fällen sich der Portierungsaufwand rechtfertigt. Anschließend werden in Kapitel 7 die erreichten Ziele zusammengefasst und ein Ausblick für zukünftige Untersuchungen gegeben.

2 FPGAs

Ein **F**ield **P**rogrammable **G**ate **A**rray (FPGA) ist ein Halbleiterbaustein aus der Familie der „**P**rogrammable **L**ogic **D**evices“ (PLDs), der durch den Nutzer bzw. Hardware-Entwickler nach der Herstellung konfiguriert werden kann. Um zu spezifizieren, wie der Schaltkreis arbeiten soll, kommen Schaltpläne oder Hardwarebeschreibungssprachen (HDLs) zum Einsatz. Jede Funktion, die ein anwendungsspezifischer integrierter Schaltkreis (ASIC) aufweisen kann, ist auch in einem FPGA umsetzbar. Grundlagenwissen über FPGAs wird in [Wan98] vermittelt.

Wie Mikroprozessoren folgen FPGAs dem mooreschen Gesetz (engl. Moore's Law) und sind mit der Zeit schneller, komplexer und in ihrer Strukturbreite kleiner geworden. Während FPGAs früher auf einfache „Glue Logic“ beschränkt waren, sind sie heute der Umsetzung von sog. Ein-Chip-Systemen (SoCs) gewachsen. FPGAs mit über einer Million Gatter sind derzeit keine Besonderheit mehr und ermöglichen völlig neue Anwendungen. Zusätzlich integrieren sie übliche Fähigkeiten von anwendungsspezifischen Standardprodukten (ASSP), wie z.B. PCI Express, USB, usw. Damit stellt die programmierbare Logik einen wesentlichen Beitrag zur Technologie der Zukunft dar.

In diesem Kapitel werden zunächst einige Beispiele für die verschiedensten Anwendungsgebiete von FPGAs gegeben. Anschließend wird deren grundlegende Architektur beschrieben und auf die Vorteile und die Schwachpunkte bei der Verwendung programmierbarer Logik eingegangen. Die in dieser Arbeit zum Einsatz kommenden Virtex-4 FPGAs und ihre Besonderheiten werden ebenso näher betrachtet. Auf die Programmiermöglichkeiten von FPGAs wird in Kapitel 4 am Beispiel der RASC-Plattform genauer eingegangen und inwiefern vorgefertigte Designs (IP-Cores) und höhere parallele Programmiersprachen (siehe z.B. 4.1.1) die Nutzbarkeit von FPGAs verbessern und Anwendungsportierung in Hardware erleichtern. Schließlich wird der Einsatz von FPGAs als Hardware-Beschleuniger im Hochleistungsrechnen erläutert.

2.1 Typische Einsatzgebiete

FPGAs haben sich bis heute besonders im Bereich der eingebetteten Systeme etabliert. Durch ihre Rekonfigurierbarkeit bieten sie weitreichende Einsatzmöglichkeiten in verschiedenen Bereichen und An-

wendungen. Besonders in Bereichen, in denen Algorithmen oder Protokolle schnell weiterentwickelt werden, ist die Verwendung rekonfigurierbarer Lösungen vorteilhaft. Produkte können so schneller auf den Markt gebracht und an neue Entwicklungen angepasst werden. Außerdem lassen sich Entwicklungsfehler durch die Rekonfigurierbarkeit nachträglich beheben. Im Folgenden werden einige Beispiele für den Einsatz von FPGAs genannt (vgl. [Xil]).

Prototypen: FPGA-basierte Prototypen dienen der Verifikation eines Hardwareentwurfs. Da die Umsetzung eines Designs auf einen ASIC sehr kostenintensiv ist, kann die Funktionsfähigkeit der entwickelten Hardware vorher auf einem FPGA überprüft werden. Zudem kann die (Gesamtsystem-) Simulation einer komplexen Schaltung nicht mehr effizient durchgeführt werden.

Automobilindustrie: Die Automobilelektronik nutzt zunehmend FPGA-basierte Schaltungen, um z.B. den effizienten Transport von Datenströmen zu steuern, mehrere Peripheriefunktionen im Auto zu vernetzen, Navigation, Infotainment oder Fahrerassistenzsysteme anzusteuern.

Signalverarbeitung: Für Lösungen zur Übertragung von Video- und Audio-Daten vom Erzeuger zum Verbraucher und die entsprechende Signalverarbeitung, wie z.B. die Decodierung von Musik, Sprache und Datendiensten werden FPGAs eingesetzt.

Kommunikationstechnik: FPGAs sind sowohl in der drahtlosen, als auch in der drahtgebundenen Kommunikation einsetzbar. Sie werden beispielsweise bei der Protokoll-Abarbeitung für verschiedene Übertragungsstandards, wie GPRS oder Ethernet-MAC-Layer verwendet.

Luft- und Raumfahrt/militärische Anwendungen: In diesem Bereich werden häufig FPGAs mit strahlungsfesten Gehäusen eingesetzt, um mit IP-Cores Bild- und Tonsignale zu verarbeiten. Ein weiterer Nutzungsgrund im militärischen Bereich ist das Verlorengehen der Programmierung des FPGAs, wenn kein Strom anliegt und damit die Sicherstellung, dass Informationen geheim bleiben.

Verbraucher: Auch für die Verbraucherelektronik bieten FPGAs kostengünstige Lösungen. Sie werden beispielsweise in digitalen Flachbildschirmen, für das Heimnetzwerk oder in Digitalempfängern und SetTop-Boxen verwendet.

Industrie/Forschung/Medizin: Marktspezifische angepasste Lösungen, wie z.B. Motorkontrolle in der Automatisierung, High-End-Bildverarbeitung in der Medizin oder Erkennung von Mustern in der Biologie, spielen in diesen Bereichen eine wesentliche Rolle.

Speicher- und Archivierungssysteme: FPGAs eignen sich für die Realisierung schneller Speicher- und Schnittstellensysteme. Sie werden unter anderem in NAS- und SAN-Systemen verwendet.

Beschleunigung von Algorithmen: Mit dieser Art der Anwendung von FPGAs beschäftigt sich diese Diplomarbeit. Die rekonfigurierbare Hardware fungiert dabei als Spezial- oder Koprozessor.

Rechenintensive Teile eines Algorithmus werden nicht mehr auf der CPU, sondern auf einem oder mehreren FPGAs ausgeführt. In Abschnitt 2.6 wird auf die Funktion von FPGAs als Hardware-Beschleuniger im HPC genauer eingegangen.

2.2 Aufbau und Grundstruktur / Architektur

Üblicherweise sind FPGAs aus einer regelmäßigen Struktur von konfigurierbaren Logikblöcken (CLBs), I/O-Ports und Verbindungskanälen, wie Abbildung 2.1 zeigt, aufgebaut.

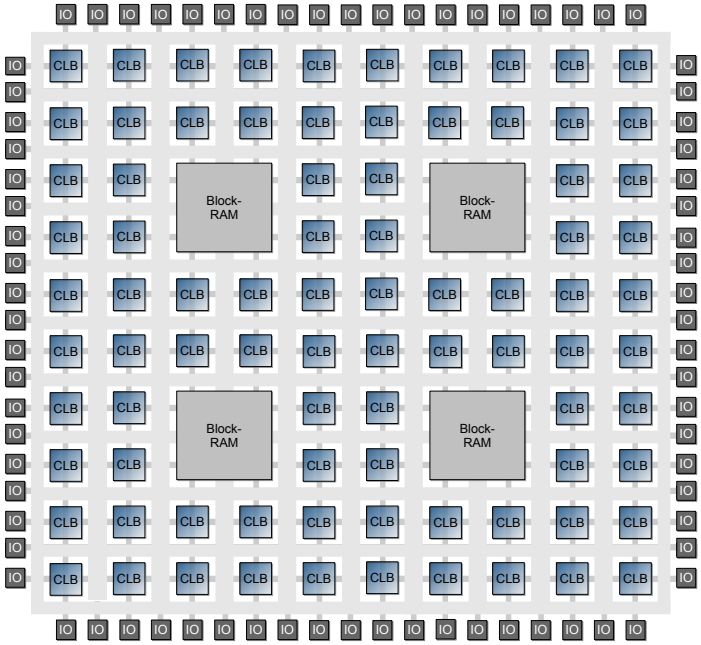
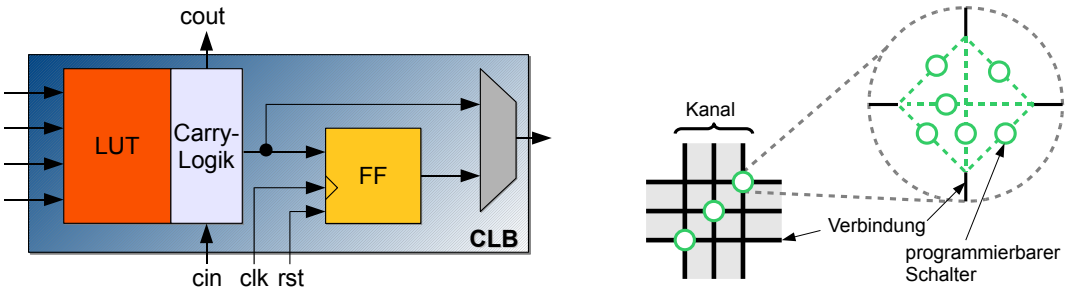


Abbildung 2.1: Typische Architektur eines FPGA

Um eine Schaltung auf einen FPGA abbilden zu können, muss dieser ausreichend geeignete Ressourcen bereitstellen. Während die Anzahl der benötigten CLBs und I/O-Ports einfach aus dem Design heraus bestimmt werden kann, variiert die Anzahl der Routing-Elemente sogar bei Designs mit gleicher Menge an Logik. Beispielsweise benötigt ein Koppelfeld (engl. crossbar switch) mehr Verbindungsressourcen als ein systolisches Array mit der selben Anzahl von Gattern. Da ungenutzte Verbindungsressourcen die Kosten erhöhen und FPGA-Fläche belegen, werden nur so viele Verbindungskanäle implementiert, wie benötigt werden, um die meisten Designs routen zu können. Dies führt jedoch dazu, dass Schaltungen, die von der Anzahl der benötigten Logik in einen FPGA passen würden, u.U. nicht geroutet werden können.

Der typische FPGA-Logikblock besteht aus einer Lookup-Tabelle (LUT) mit vier Eingängen und einem Flipflop, wie Abbildung 2.2a zeigt. Die meisten FPGAs besitzen zudem noch Carry-Logik, die schnelle

Verbindungen zwischen mehreren CLBs und damit z.B. auch schnelle Additionen mit großer Bitbreite ermöglicht. Eine 4-Eingangs-LUT hat eine Kapazität von 16 verschiedenen Kombinationen. Die in aktuellen FPGAs verwendete 6-Eingangs-LUT erhöht die Kapazität auf 64 verschiedene Kombinationen. Soll beispielsweise ein Paritätsbit für einen 32 Bit breiten Bus (siehe Abbildung 2.3) berechnet werden, sind 32 Eingänge über XOR-Verknüpfungen mit einem Ausgang zu verbinden. Wird diese Operation mit einer 4-Eingangs-LUT durchgeführt, werden elf LUTs und drei Logikebenen benötigt. Mit einer 6-Eingangs-LUT lässt sich dieselbe Funktion mit nur sieben LUTs und zwei Logikebenen ausführen. Neben der Ressourceneinsparung durch die Verwendung von weniger LUTs, wird auch der kritische Pfad kürzer und die benötigten Verdrahtungsressourcen verringern sich. Vorteile bieten sich dadurch primär bei der Verwendung von Operationen mit großer Bitbreite.



(a) Einfacher konfigurierbarer Logikblock

(b) Schaltmatrix

Abbildung 2.2: FPGA Logikblock und Schaltmatrix (vgl. [Wik09])

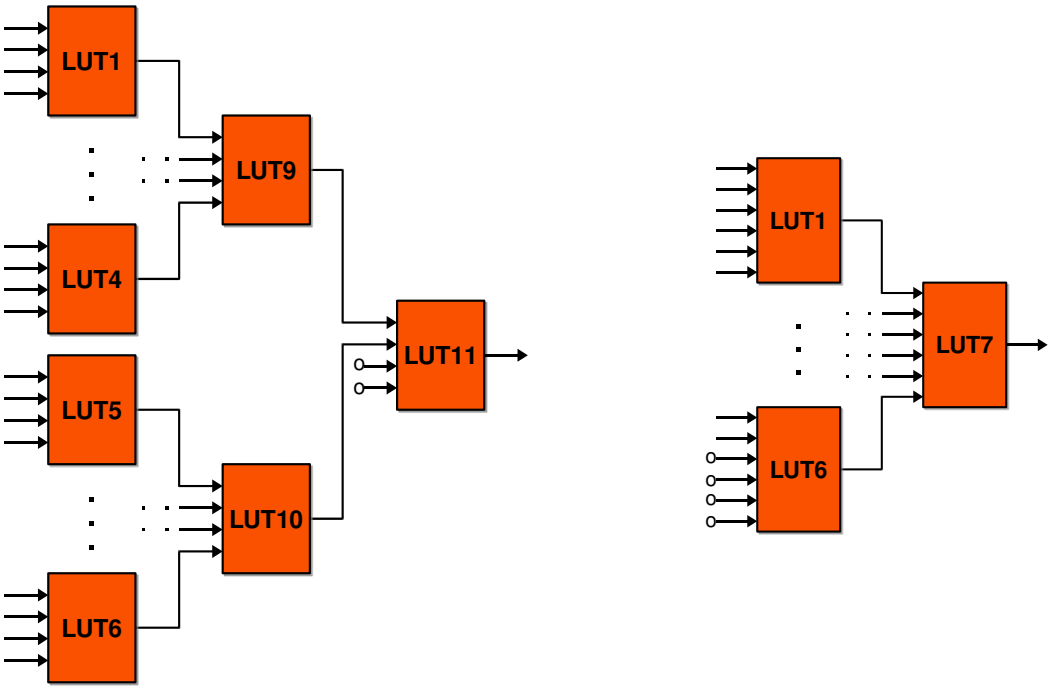


Abbildung 2.3: Paritätsbit für 32-Bit-Wert: 4-Eingangs-LUT (links), 6-Eingangs-LUT (rechts)

Ein CLB hat pro LUT nur einen Ausgang, an welchem entweder der Wert des Flipflop- oder LUT-Ausgangs anliegt. Die Durchschaltung des LUT-Ausgangs erzeugt Logik, während ein Flipflop Werte puffert und damit den kritischen Pfad unterbricht. Neben den Eingängen für die LUTs haben CLBs noch weitere Eingänge wie z.B. für das Takt-Signal. Dieses und andere Signale mit hohem Fanout werden normalerweise über spezielle Verbindungsressourcen verteilt. Weitere spezielle Verbindungsressourcen zwischen benachbarten Logikblöcken können sog. „Carry-Chains“ implementieren.

CLBs sind untereinander über Schaltmatrizen und entsprechende Verbindungskanäle gekoppelt. In Abbildung 2.2b wird eine solche Schaltmatrix mit programmierbaren Schaltern gezeigt. Für diesen Fall einer planaren Topologie kann eine Leitung mit drei ebenso an die Schaltmatrix gekoppelten Kanälen verbunden werden, allerdings eine bestimmte Leitung auch nur auf eine andere Leitung in einem benachbarten Kanal umgeleitet werden (z.B. wird Leitung 1 mit Leitung 1 aus einem anderem Kanal verbunden, Leitung 2 mit Leitung 2 aus einem anderen Kanal, usw.).

In modernen FPGAs (siehe Abschnitt 2.4) werden fest verdrahtete Schaltungen mit spezieller Funktionalität integriert. Diese benötigen weniger Chip-Fläche und können schneller getaktet werden als äquivalente Schaltungen, welche durch normale CLBs abbgebildet werden.

2.3 Stärken und Schwächen von FPGAs

Zur Darstellung einiger Stärken und Schwächen von FPGAs werden in diesem Abschnitt fest verdrahtete Chips, wie ASICs, Mikroprozessoren und andere Hardware-Beschleuniger, als Vergleich herangezogen. Prinzipiell lässt sich feststellen, dass FPGAs die Flexibilität von Software und die Leistungsfähigkeit dedizierter Hardware vereinen.

Eine hohe Verarbeitungsleistung durch FPGAs ist immer dann zu erwarten, wenn die Anwendung im Wesentlichen mit Integerverarbeitung und Logikoperationen auskommt und zudem gut parallelisierbar ist. Im Vergleich zur parallelen Abarbeitung von Software auf Mehrkernprozessoren oder Rechnersystemen mit vielen CPUs, wird bei FPGAs von Hardwareparallelität gesprochen. Sequentielle Algorithmen, die auch durch Pipelining keine Parallelverarbeitung ermöglichen, werden kaum Geschwindigkeitsgewinnen erreichen. Dies liegt zum einen an den Taktraten von FPGAs, die ungefähr um den Faktor zehn geringer sind als bei Mikroprozessoren und zum anderen am zusätzlichen Kommunikationsaufwand.

Während Prozessoren eine feste Verarbeitungsbreite aufweisen, kann der Hardware-Entwickler die Bitbreite von Datentypen und Operationen bestimmen. Dies ist einer der entscheidenden Vorteile von FPGAs, welcher die Anpassung der Hardware an einen Algorithmus auf Bit-Ebene ermöglicht und damit dazu beiträgt, Ressourcen einzusparen.

Ein verhältnismäßig schwacher Punkt bei FPGAs ist derzeit noch die Verarbeitung von Fließkommawerten (siehe Abschnitt 2.5), obwohl darauf spezialisierte FPGAs mit vielen DSP-Blöcken durchaus gute Fließkommaverarbeitungsleistung erbringen können. Für Algorithmen, die primär auf solchen Datentypen arbeiten, sind andere Beschleuniger, wie z.B. ClearSpeed- oder Grafikkarten, meist besser geeignet und bringen auch bessere Performance.

Der wesentliche Vorteil von FPGAs gegenüber ASICs ist die Rekonfigurierbarkeit, welche es ermöglicht, eine an bestimmte Anforderungen bereits angepasste Hardware im Nachhinein noch zu modifizieren oder auch zu erweitern. Damit bieten FPGAs eine ähnliche Flexibilität wie Software. Diese Eigenschaft wird u.a. dafür genutzt Prototypen zu erstellen, um Ideen und Konzepte bereits in Hardware testen zu können. Für einen Langzeiteinsatz ist die Rekonfigurierbarkeit ebenfalls von Vorteil, da sich über die Jahre Spezifikationen verändern und eine Anpassung der FPGAs ohne erneute Fertigung und Austausch der Hardware möglich ist. Damit reduzieren sich u.U. Wartungskosten. Eine weitere Eigenschaft moderner SRAM-basierter FPGAs, dynamische partielle Rekonfiguration[1], ist zunehmend auch für Anwendungen in der Praxis interessant (z.B. bei Fahrerassistenzsystemen in der Automobilindustrie). Sie ermöglicht es Anwendungen sich an wechselnde Bedingungen anpassen zu können, ohne dass der komplette FPGA mit einem neuen Bitstream rekonfiguriert werden muss.

Auf FPGAs basierende Lösungen ermöglichen es mitunter Markteinführungszeiten stark zu verkürzen, da das Silizium bereits geprüft ist und kein langer Fertigungsprozess notwendig ist. Der kostenintensive Entwicklungs- und Fertigungsprozess von ASICs führt dazu, dass FPGAs für kleine und mittlere Stückzahlen kostengünstiger sind. Ab größeren Stückzahlen macht sich der höhere Einzelpreis von FPGAs jedoch bemerkbar.

Im Vergleich zu ASICs haben FPGAs einen höheren Leistungsbedarf und eine geringere Logikdichte (ca. zehnfacher Flächenbedarf bei gleicher Technologie). Wird jedoch der Leistungsbedarf moderner CPUs betrachtet, sind FPGAs deutlich sparsamer. Insbesondere bei einem auch im HPC an Relevanz gewinnenden Maß „Performance pro Watt“ sind FPGAs bei geeigneter Algorithmen-Wahl weitaus effizienter. Allerdings stehen sie hier in direkter Konkurrenz zu anderen Hardware-Beschleunigern (z.B. ClearSpeed, GPGPUs), welche sehr gute Werte hinsichtlich dem Verhältnis von Performance zu Leistungsaufnahme aufweisen.

Fest verdrahtete Logik wie ASICs und Mikroprozessoren ermöglichen höhere Taktfrequenzen als FPGAs. Aktuell sind FPGAs mit bis zu 600 MHz verfügbar, typisch sind jedoch 20 bis 250 MHz. Die Flexibilität von modernen Mikroprozessoren (Befehlssatzerweiterungen, etc.) und FPGAs ziehen Nachteile gegenüber ASICs mit sich. Da Aufgaben nicht, wie bei einem spezialisierten Baustein, optimal abgearbeitet

[1] Rekonfiguration von Teilen des FPGAs zur Laufzeit (während andere Blöcke weiter arbeiten)

werden können, sind Energieverbrauch, Datendurchsatz, Chip-Fläche, Taktfrequenz und andere Zielparameter schlechter.

Ein mitunter entscheidender Punkt, der gegen die Verwendung von FPGAs sprechen kann, ist der Entwicklungsaufwand, der beim Hardware-Entwurf betrieben werden muss (siehe Abschnitt 6.2.1 und 6.2.1). Zudem sind die Preise moderner High-End-FPGAs sehr hoch. Nicht zu vergessen ist, dass auch FPGAs in ihrer Ausstattung (z.B. eingebetteter Speicher, analoge Elemente, I/O-Ressouren) beschränkt sind.

SRAM-basierte FPGAs müssen zudem bei jedem Systemstart konfiguriert werden. Es sind also zusätzliche externe Komponenten (z.B. EEPROM oder Flash-Speicher und Mikrokontroller) notwendig, um den Bitstream in den FPGA zu laden. Das bedeutet auch, dass die Funktionalität eines FPGAs nicht direkt nach dem Einschalten zur Verfügung steht, sondern erst nach dem Laden. Dies kann je nach eingesetzter Technik einige Zeit, typischerweise im Bereich von Millisekunden, dauern.

2.4 Virtex-4 FPGAs

Die Virtex-4 FPGA-Familie umfasst drei Plattformen, welche für verschiedene Anwendungszwecke konzipiert sind: LX - Logik, FX - Features, SX - Signalverarbeitung. Die Unterschiede liegen in der Anzahl der frei programmierbaren Logikzellen, den integrierten fest verdrahteten Spezialblöcken und der Menge an digitalen Signalverarbeitungsblöcken (DSP-Slices). Zudem besitzt jede Plattform FPGA-Modelle in verschiedener Größe und Ausführung. Die in dieser Arbeit verwendeten Virtex-4-LX-FPGAs beinhalten die meisten frei programmierbaren Logikzellen der gesamten Virtex-4 FPGA-Familie.

Xilinx untergliedert die Virtex-4-CLBs in vier sog. Slices (vgl. Abbildungen 2.4 und 2.5), wobei zwei davon ausschließlich zur Umsetzung von Logik dienen, während die anderen Beiden außerdem als verteilter Speicher oder Schieberegister verwendet werden können. Die Slices bestehen jeweils aus zwei 4-Input-LUTs und zwei Flipflops, womit die Verarbeitungsbreite eines CLBs auf maximal 64 Bit beschränkt ist.

Die neueste Generation von Xilinx-FPGAs, die Virtex-6-Serie, hat CLBs bestehend aus zwei Slices mit jeweils vier 6-Input-LUTs und vier Flipflops (siehe [Xil09b]), was für heutige Anwendungen von High-End-FPGAs eine effizientere Aufteilung sein soll. Virtex-4-Chips werden aus 300 mm Wafern in 90-nm Technologie produziert, während neuere Virtex-6-FPGAs bereits in 40-nm Technologie hergestellt werden. Durch ein effizienteres Verbindungsnetzwerk und kleinere Strukturbreiten kann damit bei gleicher Taktfrequenz die Leistungsaufnahme der FPGAs verringert werden.

Die Speicherressourcen von Virtex-4 FPGAs reichen von 48 bis 552 18Kbit Dual-Port-RAM-Blöcken (500 MHz) und von 86 bis 1392 KBits als verteilten Speicher (Verwendung von CLBs). Zudem sind

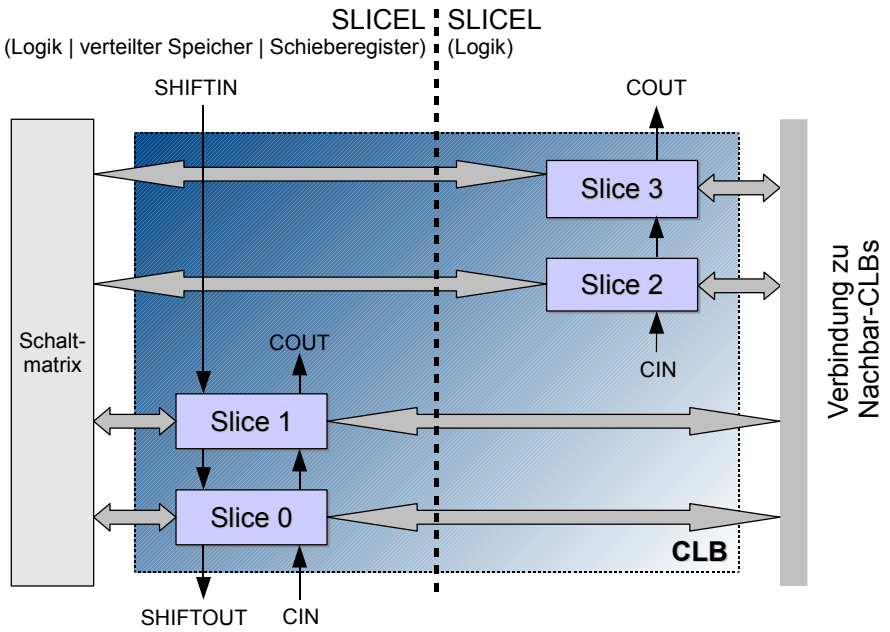


Abbildung 2.4: Konfigurierbarer Logikblock (CLB) der Virtex-4 FPGAs (vgl. [Xil08c])

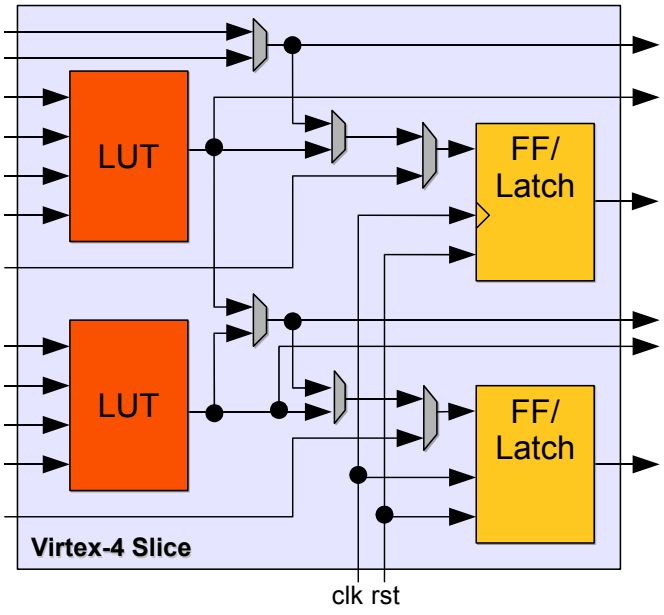


Abbildung 2.5: Stark vereinfachter Virtex-4-Slice (ohne Carry- und Speicher-Logik, vgl. [Xil08c])

zwischen 32 und 512 „Xtreme DSP Slices[2]“ (500 MHz) und 4 bis 32 „**D**igital **C**lock **M**anager“ (DCM) Blöcke integriert. Durch die unterschiedlich langen Verbindungen erreicht ein Taktsignal die einzelnen CLBs nicht gleichzeitig. Für Pipelines z.B. ist es jedoch wichtig, dass sie synchron getaktet sind, da sonst undefinierte Werte zustande kommen können. Mit Hilfe von „Delay-Locked Loops“ (DLLs) sorgen die DCMs dafür, dass der Takt alle CLBs gleichzeitig erreicht.

Außerdem gibt es für die FX-Plattform sog. Hard-IP-Blöcke (fest verdrahtete Schaltungen), wie z.B. PowerPC-Prozessor-Blöcke (bis 450MHz, maximal zwei pro FPGA), Ethernet MACs (zwei oder vier pro FPGA) und RocketIO Multi-Gigabit-Transceiver(MGT)-Blöcke[3] (8 bis 24 pro FPGA). Fest verdrahtete Logik für Multiplizierer, Prozessorkerne etc. benötigt weniger Fläche auf dem Chip und kann schneller getaktet werden als logisch gleichartige Schaltungen unter der Verwendung von CLBs.

Virtex-4 FPGAs bieten mit dem Dynamic Reconfiguration Port (DRP) eine einfache Möglichkeit, ähnlich dem Block-RAM-Interface, die Funktionalität einer Anwendung während der Laufzeit zu ändern. Dieser ist in zwei Arten von Funktionsblöcken integriert: DCMs und MGTs.

Detailliertere Informationen zu allen FPGAs der Virtex-4-Serie können der Produktspezifikation [Xil07] entnommen werden. Auf die Virtex-4-Architektur und integrierte Funktionsblöcken wie Block-RAM, DCMs und E/A-Ressourcen wird im Benutzerhandbuch [Xil08c] eingegangen, während [Xil08b] die Konfigurationsmöglichkeiten (Varianten den Bitstream in den FPGA zu laden, zur Verfügung stehende Schnittstellen, Aufbau des Bitstreams) beschreibt.

2.5 Fließkommaverarbeitungsleistung

Im Hochleistungsrechnen (engl. High Performance Computing - HPC) ist die Verarbeitungsleistung von Gleitkommazahlen ein wesentliches Kriterium, um die Leistungsfähigkeit eines Systems zu bewerten. Als Maßeinheit werden Fließkommaoperationen pro Sekunde (FLOP/s) verwendet. In diesem Abschnitt wird Gleitkomma oder Fließkomma mit der aus dem Englischen kommenden Bezeichnung FP (floating point) abgekürzt.

Auch wenn die Stärke von FPGAs in anderen Bereichen liegt, soll die zu erwartende FP-Verarbeitungsleistung näher untersucht werden. Als Referenz dienen wie üblich Mikroprozessoren, wobei auch Grafikprozessoren (GPUs) als Vergleich herangezogen wurden. In [Str07] wird die FP-Performance von FPGAs anhand theoretischer Berechnungen abgeschätzt. Eine Aktualisierung und Anpassung dieser Leistungsbewertung unter Berücksichtigung von [KC07] wird in [SSWW08] vorgenommen.

[2] Jeder Xtreme DSP Slice besteht aus einem 18x18 Multiplizierer, einem Addierer und einem Akkumulator
[3] serielle Transceiver mit 622 Mb/s bis 6.5 Gb/s Baudrate

FP-Verarbeitung findet typischerweise in der verallgemeinerten Matrix-Multiplikation (engl. General Matrix Multiply - GEMM) Verwendung. Um maximale Performance für Mikroprozessoren zu erreichen, werden vom Programmierer hochoptimierte GEMM-Funktionen (für einfache Genauigkeit (32 Bit) SGEMM und doppelte Genauigkeit (64 Bit) DGEMM) verwendet und Zeiger auf die zu multiplizierenden Matrizen übergeben. Bei FPGAs steht dem Programmierer im Optimalfall eine vom Systemanbieter mitgelieferte Routine zur Verfügung, welche dieselben Zeiger akzeptiert. Im ersten Fall wird die GEMM-Funktion auf einem Mikroprozessor ausgeführt und in dessen Speicher gearbeitet. Im zweiten Fall wird der Mikroprozessor mittels direktem Speicherzugriff (DMA) Daten an den FPGA oder ihm zur Verfügung stehende Speicherressourcen übertragen. Die durch die rekonfigurierbare Logik berechneten Ergebnisse werden anschließend wieder in den Speicher des Mikroprozessors übertragen.

Auch wenn die Bandbreite und Latenz der Kommunikation zwischen Mikroprozessor und FPGA mitunter starken Einfluss auf die Performance der Anwendung hat, wird sie für diese theoretische Betrachtung vernachlässigt. Die maximal von einem Mikroprozessor erreichbaren FLOP/s errechnen sich aus den pro Takt und Kern durchführbaren FP-Operationen, multipliziert mit der Taktfrequenz und der Anzahl der Kerne auf dem Prozessor. Damit kann ein Vierkern Opteron Prozessor mit 2,5 GHz theoretisch 40 GFLOP/s (*4 Operationen/Takt · 4 Kerne · 2,5 GHz*) bei doppelter Genauigkeit (64 Bit) und 80 GFLOP/s bei einfacher Genauigkeit (32 Bit) erreichen.

Ein FPGA besitzt weder FP-Addierer noch FP-Multiplizierer, sondern generische Logik, welche vom Nutzer konfiguriert werden kann. Um also eine vergleichbare maximale 64-Bit-FP-Verarbeitungsleistung zu bestimmen, muss herausgefunden werden, wieviele Addierer und Multiplizierer bei welcher Taktfrequenz auf dem FPGA implementierbar sind. Die Spitzenleistung berechnet sich dann aus der durch den FPGA zur Verfügung stehenden Logik, dividiert durch die von einer FP-Verarbeitungseinheiten benötigten Logik, multipliziert mit der maximal erreichbaren Taktfrequenz des Designs.

Berechnungsgrundlagen (FPGA)

Als Basis der folgenden Berechnungen dienen vier Dokumente von Xilinx: die Kenndatenübersichten der Virtex-4-, Virtex-5- und Virtex-6-Familie ([Xil07], [Xil09a], [Xil09b]) und die Spezifikation des „Xilinx Floating-Point Core“ ([Xil08a]). Aus letzterem können die zur Implementierung von FP-Verarbeitungseinheiten benötigten Ressourcen für Virtex-4- und Virtex-5-FPGAs entnommen werden. Einige Kenndaten, über die in diesem Abschnitt betrachteten FPGAs, sind in Tabelle 2.1 zusammengefasst und werden in den folgenden Berechnungen verwendet. Zu beachten ist außerdem, dass die Slices der Virtex-Familien unterschiedlich viele FFs und LUTs beinhalten.

Um die FP-Verarbeitungsleistung von FPGAs realistisch abschätzen zu können, muss ein Teil der Lo-

FPGA	Slices	LUTs	FFs	DSP-Slices	FP-LUTs	FP-FFs
Virtex-4 LX200	89.088	178.176	178.176	96	105.451	105.451
Virtex-4 SX55	24.576	49.152	49.152	512	19.435	19.435
Virtex-5 LX330	51.840	207.360	207.360	192	124.907	124.907
Virtex-5 SX240T	37.440	74.880	74.880	1056	86.507	86.507
Virtex-6 LX760	118.560	474.240	948.480	864	302.827	618.987
Virtex-6 SX475T	74.400	297.600	595.200	2016	185.067	383.467

Virtex-4-Slice: 2 FFs und 2 LUTs; Virtex-5-Slice: je 4 FFs und 4 LUTs; Virtex-6-Slice: 8 FFs und 4 LUTs

Tabelle 2.1: Kenndaten einiger Virtex-FPGAs

gik für die Host-FPGA-Kommunikationsschnittstelle und den Speicherkontroller reserviert werden. In [SSWW08] werden dafür 20.000 Flipflops (FFs) und 20.000 Lookup-Tabelle (LUTs) veranschlagt und in folgenden Berechnungen verwendet. Für die RASC Core Services liegt dieser Wert je nach Kommunikationsvariante zwischen 8.000 und 17.000 FFs und 7.000 und 13.000 LUTs (vgl. Abschnitte 5.1.1, 5.2.1 und 5.3.1). Da es möglich ist Kommunikationsvarianten zu kombinieren, kann auch eine größere Menge an Ressourcen durch die Core Services belegt werden. Die Gesamtzahl an LUTs und FFs wird zusätzlich um 33% reduziert (siehe [KC07]), um das Routing des Designs zu ermöglichen. Damit stehen für FP-Operationen nur $\frac{2}{3} \cdot (FFs - 20.000)$ FFs und $\frac{2}{3} \cdot (LUTs - 20.000)$ LUTs zur Verfügung (vgl. grau hinterlegte Spalten aus Tabelle 2.1). Schließlich wird noch eine um 15% verringerte maximale Taktfrequenz der FP-Schaltungen (aus [Xil08a]) angenommen, um Verdrahtungsanforderungen gerecht zu werden.

Performance-Abschätzung

Mit dem „Xilinx Floating-Point Core“ können für verschiedene FPGAs zusätzlich noch verschiedene Implementierungen (vier für Multiplikation, zwei für Addition) vorgenommen werden, welche sich hinsichtlich der Nutzung von DSP-Slices, Logik und Flipflops und der maximal erreichbaren Taktfrequenz unterscheiden. So sind FP-Operationen ausschließlich mit CLBs oder unter zusätzlicher Verwendung von DSP-Slices umgesetzbar. Werden weniger DSP-Slices verwendet, ist der Bedarf an FFs und LUTs für die gleiche Anzahl an FP-Verarbeitungseinheiten höher und die maximale Taktfrequenz niedriger. In [SSWW08] werden die besten Resultate aller möglichen Kombinationen von Implementierungen, bezogen auf das Verhältnis von Addition und Multiplikation, ermittelt und zudem die Speedups zum 2,5 GHz Vierkern Opteron Prozessor angegeben. Tabelle 2.2 zeigt einen Ausschnitt der Ergebnisse dieser theoretischen FPGA-FP-Betrachtung.

Nach diesen Werten ist der Virtex-5 SX240T bei jedem beliebigen Verhältnis von Multiplikation und Addition schneller als der Opteron Prozessor. Besonders auffällig ist, dass sich der FPGA-Speedup mit grö-

Verhältnis	GFLOP/s (64 Bits / 32 Bits)			Speedup over Opteron	
Add:Mult	Opteron	LX330	SX240T	LX330	SX240T
nur Add.	17,00 / 34,00	32,96 / 85,09	29,97 / 80,39	1,94 / 2,50	1,76 / 2,36
8:1	19,13 / 38,25	33,77 / 87,43	29,97 / 84,56	1,77 / 2,29	1,57 / 2,21
4:1	21,25 / 42,50	28,14 / 90,45	32,10 / 92,22	1,32 / 2,13	1,51 / 2,17
2:1	25,50 / 51,00	21,71 / 94,47	34,97 / 104,40	0,85 / 1,85	1,37 / 2,05
1:1	34,00 / 68,00	18,89 / 89,04	39,29 / 122,50	0,56 / 1,31	1,16 / 1,80
1:2	25,50 / 51,00	16,28 / 88,72	42,37 / 141,98	0,64 / 1,74	1,66 / 2,78
1:4	21,25 / 42,50	15,07 / 81,81	43,33 / 149,64	0,71 / 1,92	2,04 / 3,52
1:8	19,13 / 38,25	14,47 / 79,08	40,45 / 153,47	0,76 / 2,07	2,12 / 4,01
nur Mult.	17,00 / 34,00	13,47 / 79,69	37,56 / 162,52	0,79 / 2,34	2,21 / 4,78
Optimal	-	34,30 / 94,47	44,94 / 162,52	-	-

Tabelle 2.2: FP-Performance von Virtex-5-FPGAs im Vergleich zu Opteron Prozessor

ßerer Entfernung vom Verhältnis 1:1 verbessert und im Optimalfall für den SX240T 4,78 erreicht. Tabelle 2.3 zeigt einen Überblick der Top-FPGA-Modelle von Xilinx aus den letzten drei Virtex-Generationen und ihre FP-Performance bei Verwendung von genauso vielen Multiplizierern wie Addierern. Damit ist ein direkter Vergleich mit Mikroprozessoren, welche Multiplizier-Addier-Einheiten verwenden möglich. Die Werte wurden hier von Hand berechnet und müssen damit nicht dem Optimum entsprechen, sollten aber nicht weit davon entfernt sein.

FPGA	Mult-Add	Taktfrequenz	GFLOP/s
Virtex-4 LX200	6+25 / 90+0	153,85 / 255,00 MHz	9,54 / 45,90
Virtex-4 SX55	10+ 0 / 26+0	277,95 / 331,50 MHz	5,56 / 17,24
Virtex-5 LX330	21+26 / 133+0	201,45 / 348,50 MHz	18,93 / 42,85
Virtex-5 SX240T	66+ 0 / 189+0	337,45 / 348,50 MHz	44,54 / 131,73
Virtex-6 LX760	96+60 / 432+0	219,76 / 380,18 MHz	68,57 / 328,48
Virtex-6 SX475T	163+ 0 / 504+0	341,24 / 380,18 MHz	111,24 / 383,22

Tabelle 2.3: Virtex-FPGAs: FP-Performance Multiplikation-Addition (64 Bit / 32 Bit)

Die zweite Spalte gibt an, wieviele Multiplizier-Addier-Einheiten auschließlich in Logik und mit DSP-Slices implementiert wurden (DSP + Logik = FP-Einheiten). Die in [Xil08a] angegebenen maximalen Taktfrequenzen wurden um 15% verringert und sind in Spalte drei eingetragen. Da die für die Virtex-6-FPGAs benötigten Ressourcen zur Implementierung von FP-Verarbeitungseinheiten noch nicht angegeben sind, wurden die Werte der Virtex-5 angenommen. Die maximale Taktfrequenz wurde jedoch mit $\frac{12}{11}$ multipliziert, weil Virtex-6-DSP-Slices anstelle von 550 MHz mit 600 MHz arbeiten. Die FP-Performance errechnet sich aus dem mit zwei multiplizierten Produkt der Spalten fünf und sechs, da eine Multiplizier-Addier-Einheit gleichzeitig Multiplikation und Addition durchführen kann.

Der begrenzende Faktor für die Signalverarbeitungs-FPGAs ist meist der Logik- oder FF-Bedarf, während bei auf Logik optimierten FPGAs (LX) die DSP-Slices nicht ausreichen, damit FP-Einheiten in

CLBs umgesetzt werden und die Taktfrequenz verringert werden muss. Virtex-6-FPGAs beinhalten pro Slice acht FFs und vier LUTs, wodurch der FF-Bedarf kein die FP-Performance begrenzender Faktor ist. Die schwache Leistung des Virtex-4 SX55 liegt an der geringen Anzahl an FFs und LUTs, wodurch bei voller Nutzung der DSP-Slices nur etwa 37% der insgesamt verfügbaren DSP-Slices verwendet werden können.

Verglichen mit GPUs (siehe [Wag08]) ist die theoretisch erreichbare FP-Performance von FPGAs deutlich geringer. NVIDIAs Tesla C870 und AMDs FireStream 9170 haben eine theoretische Maximalleistung von über 500 GFLOP/s für einfache Genauigkeit (32 Bit). Bei Messungen konnten jedoch maximal 200 GFLOP/s erreicht werden.

Wird der Fakt betrachtet, dass FPGAs eigentlich nicht für FP-Verarbeitung entwickelt wurden, bieten neue Modelle dennoch sehr gute FP-Leistung und können mit aktuellen Prozessoren mithalten. Einen weiteren Geschwindigkeitsschub kann mit FPGAs erreicht werden, wenn geringe Bitbreiten benötigt werden. Während eine 64-Bit-Addition nur etwa doppelt soviel Ressourcen wie eine 32-Bit-Addition benötigt, sind es beim Umstieg von 32-Bit auf 64-Bit Multiplikation viermal soviele Ressourcen. Bei einfacher Genauigkeit (32 Bit) erreicht auch ein Virtex-4 LX200 etwa 46 GFLOP/s, was fast fünfmal so schnell wie der Wert für doppelte Genauigkeit (64 Bit) aus Tabelle 2.3 ist. Zudem werden gute FP-Bibliotheken für Mikroprozessoren in Assembler handoptimiert, was einen ähnlichen Aufwand wie die Hardware-Programmierung mit sich bringt.

2.6 FPGAs als Hardware-Beschleuniger im HPC

Neben anderen Hardware-Beschleunigern werden im Hochleistungsrechnen (HPC) auch FPGAs eingesetzt. Ein Grund dafür ist der enorme Stromverbrauch auf herkömmlichen Mikroprozessor basierten, großen Rechnersystemen. Im Bereich eingebetteter Systeme ist der Stromverbrauch schon lange ein begrenzender Faktor, während dies erst seit Kurzem im HPC eine Rolle spielt. Abbildung 2.6 zeigt die Leistungsaufnahme von FPGAs und GPUs im Vergleich zu CPUs in Abhängigkeit vom Geschwindigkeitsgewinn (vgl. Formel 2.1). Als Referenz wird ein CPU-Kern mit einer Leistungsaufnahme von 20 Watt ($P_{CPU} = 20W$) angenommen, ein FPGA mit 25 Watt ($P_{FPGA} = 25W$) und ein GPU mit 125 Watt ($P_{GPU} = 125W$). Um also eine Energieersparnis von 50% zu erreichen, muss die durch den FPGA/GPU beschleunigte Anwendung 2,5/12,5 mal so schnell sein.

$$P(S_p) = \frac{P_{FPGA/GPU}}{P_{CPU} \cdot S_P} \tag{2.1}$$

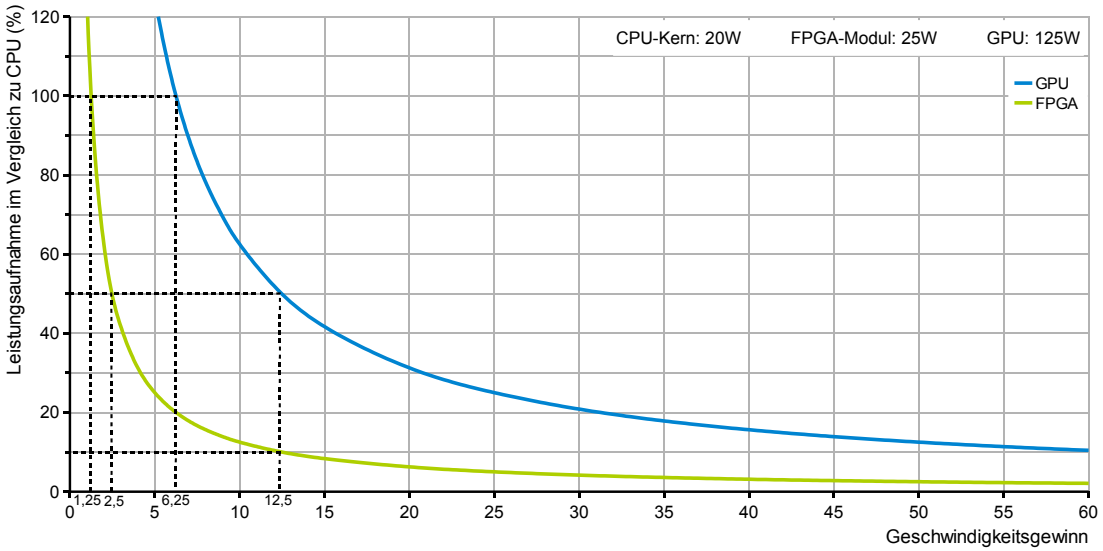


Abbildung 2.6: Energieeffizienz von FPGAs und GPUs

Diese Darstellung betrachtet jedoch nur die Recheneinheiten selber und vernachlässigt jegliche Periphrie-Komponenten, auch wenn diese zur Ansteuerung bzw. zum Betrieb notwendig sind. Es kommt also noch ein schwer bestimmbarer Faktor hinzu, welcher neben Komponenten, wie z.B. Speicher, auch die Kühlung mit einschließt.

Es werden hier ausschließlich SRAM basierte FPGAs verwendet, weil diese, unter der Bedingung rekonfigurierbar zu sein, die beste Performance aufweisen. Aktuelle High-End-FPGAs mit vielen Logikblöcken sind zwar sehr kostenintensiv, bieten jedoch viele Zusatzmerkmale und fest integrierte Schaltungen, die eine schnelle Anbindung an das Host-System erlauben. Zudem kann auf derart großen Chips nahezu jeder Algorithmus implementiert werden. In FPGA-basierten Beschleuniger-Systemen übernimmt ein Host die Ansteuerung und Programmierung der FPGAs.

In Kapitel 3 wird mit SGI RASC ein System vorgestellt, welches FPGAs aus der Virtex-4 LX Familie (siehe Abschnitt 2.4) als Hardware-Beschleuniger verwendet. Andere Systeme arbeiten mit sog. „In-Socket Accelerators“, wobei einfach ein Mikroprozessor durch einen Pin-kompatiblen FPGA ersetzt wird. Die enge Kopplung von CPU und FPGA führt zu einer geringen Latenz zwischen beiden und FPGAs haben zudem noch direkten Zugriff auf die Speicherressourcen des Systems. XtremeData’s XD2000i und XD2000F sind solche Lösungen.

3 SGI RASC

Unter „**R**ekonfigurierbares **A**nwendungs-**S**pezifisches **C**omputing“ (RASC) fasst SGI die Verarbeitung spezieller Anwendungen oder Algorithmen durch rekonfigurierbare Hardware zusammen. Im Folgenden wird das RASC-System hinsichtlich verwendeter Hardware, theoretischen Beschränkungen, zugehöriger Software und ihrer Integration in das Hostsystem SGI Altix 4700 genauer vorgestellt.

3.1 SGI Altix 4700

Der Hochleistungsrechner SGI Altix 4700 ist eine Distributed-Shared-Memory-Architektur, wobei der Hauptspeicher (und damit auch die Speicherkontroller) physisch gesehen auf die verschiedenen Knoten verteilt, aber logisch als ein großer gemeinsamer Speicher sichtbar ist. Unter der Verwendung spezieller Hardware zur Gewährleistung der Cache-Kohärenz über den verteilten Speicher, wird diese Systemarchitektur auch als „cache-coherent Non-Uniform Memory Architecture“ (ccNUMA) bezeichnet. Für den Zugriff der Prozessoren auf den gemeinsamen Hauptspeicher sind spezielle Speicherkontroller (sog. SHUBs) auf den Systemknoten zuständig. Je nachdem, ob ein Speicherzugriff auf lokale oder entfernte (sich auf einem anderen Knoten befindende) Daten erfolgt, ergeben sich unterschiedliche Zugriffszeiten und Bandbreiten. Als Betriebssystem wird SuSE Linux Enterprise Server (SLES) 10 mit dem SGI ProPack 5 verwendet.

Einzelne Systemknoten der Altix 4700 werden entweder für rechenintensive Aufgaben verwendet (Rechenknoten), stellen zusätzlichen Hauptspeicher zur Verfügung (Speicherknoten) oder es handelt sich um Ein/Ausgabe-Knoten. Alle Knotentypen sind in Form von Blades, einer flachen Bauform von Platinen mit gemeinsamer Strom- und Lüftungsversorgung, realisiert. Diese Blades werden mittels des SGI-NUMAlink4-Netzwerks zu einem Shared-Memory-System in einer Fat-Tree-Topologie zusammengeschaltet. Durch dieses modulare System von Knoten lassen sich die verschiedenen Knotentypen nach Belieben variieren, wodurch die Anpassbarkeit an spezielle Anforderungen erleichtert wird.

Ein Rechenknoten (Compute-Blade, vgl. Abbildung 3.1) besteht aus bis zu zwei Intel Itanium 2 (Montecito) Prozessoren[1] und einem SHUB. Diese Kommunikationskomponente verbindet den Prozessor mit

[1] Aufgrund der Bandbreitenbeschränkung der Speicheranbindung ist in der Altix 4700 am ZIH nur ein Socket bestückt.

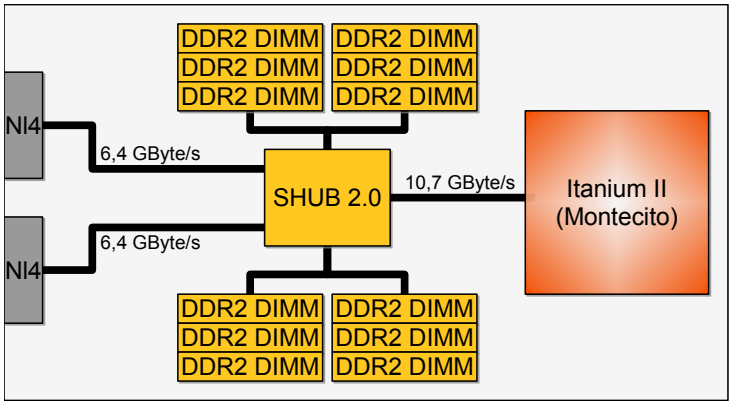


Abbildung 3.1: Altix 4700 Rechenknoten

dem physisch lokalen Hauptspeicher und stellt zwei NUMAlink4-Kanäle (je 6,4 GB/s) zur Anbindung an das Verbindungsnetzwerk bereit.

Die Intel Itanium II Doppelkernprozessoren sind mit 1,6 GHz getaktet und haben zwei Multiply-Add-Einheiten pro Core. Damit ergibt sich pro Prozessor eine Fließkomma-Spitzenleistung von 6,4 GFLOP/s (6,4 Milliarden Fließkomma-Operationen pro Sekunde). Eine CPU ist je Kern mit den in Tabelle 3.1 angegebenen großen, schnellen Caches ausgestattet, wodurch bei häufigem Cache-Zugriff eine sehr hohe Anwendungsleistung erzielt werden kann. Für die Anwendungsentwicklung ist zu beachten, dass es sich um eine IA-64 Prozessorarchitektur handelt, die aus drei Befehlen bestehende Instruktionsbündel mit einer Größe von 128 Bit verarbeitet und dass z.B. der Datentyp `long` prinzipiell 64 Bit verwendet. Mit Leistungseinbußen ist die Abarbeitung von IA-32 Anwendungen dennoch möglich.

Cache	Größe	Cache-Line	Latenz (min./typ.)
L1D	16 KB	64 Bytes	1/1 Takt
L1I	16 KB	64 Bytes	1/1 Takt
L2D	256 KB	128 Bytes	7/11 Takte
L2D	1 MB	128 Bytes	5/10 Takte
L3	9 MB	128 Bytes	14/21 Takte

Tabelle 3.1: Intel Itanium 2 Montecito[Int06] - Cache-Hierarchie

Ein-/Ausgabe-Knoten bestehen aus einer anwendungsspezifischen integrierten Schaltung (ASIC) mit der Bezeichnung TIO, welche zum einen als Cache-Kohärenzschnittstelle dient und zum anderen gängige E/A-Schnittstellen, wie zum Beispiel PCI-X oder PCI-Express, bereitstellt. Die Kohärenzschnittstelle erlaubt es, Daten cache-kohärent direkt von der E/A-Schnittstelle (z.B. PCI-X-Karte) über das NUMAlink4-Netzwerk in den verteilten Hauptspeicher auf den Prozessorknoten zu transportieren.

Durch den modularen Aufbau der Altix 4700 können auch Special-Purpose-Blades in das System integriert werden. Dies sind zum einen über PCI-Express angeschlossene Grafikkarten zur Beschleunigung

von Anwendungen durch GPUs (Graphics Processing Units) und zum anderen sog. RASC-Blades. Letztere sind mit FPGAs ausgestattet und ermöglichen eine an das Problem angepasste rekonfigurierbare Hardware zu nutzen, um Anwendungen auszuführen (siehe Abschnitt 3.2).

Die am ZIH installierte Altix 4700 besteht aus 32 Racks und verfügt insgesamt über 2048 Intel Itanium II Montecito Cores und 6,5 TB Hauptspeicher. Das System ist in fünf Partitionen aufgeteilt: eine Login-Partition, drei Rechen-Partitionen und eine interaktive Partition mit Grafik- und FPGA-Knoten. Die theoretische Spitzenleistung des Systems beträgt 13,1 TFlop/s und belegte zum Zeitpunkt der Inbetriebnahme (November 2006) mit gemessenen 11,9 TFlop/s Rang 49 unter den 500 schnellsten Rechnern weltweit.

3.2 SGI RC100 FPGA-Blade

Ein SGI RC100-Blade stellt dem Hostsystem zwei anwenderprogrammierbare FPGAs zur Verfügung und ist schematisch gesehen wie in Abbildung 3.2 aufgebaut. Es besteht neben den zwei Algorithmen-FPGAs mit je fünf 8 MByte QDR-II SRAM DIMMS[2] aus zwei TIO ASICs und einem „Loader-FPGA“.

Der Konfigurations-Bitstream wird in einem Flash-Speicher (EEPROM) gehalten und über den „Loader FPGA“, der zwischen einem PCI Port des TIO ASICs und den Algorithmen-FPGAs geschalten ist, mittels SelectMap (siehe [Xil08b]) geladen. Die Core Services (siehe Abschnitt 3.3) bilden zusammen mit dem Algorithmus den Bitstream und sind direkt mit den SSP Ports der TIO ASICs verbunden. Verwendet werden Xilinx Virtex-4-LX200-FPGAs (siehe Kapitel 2).

Ein einzelner FPGA und dessen Anbindung an das System wird als RASC-FPGA-Hardware-Modul bezeichet und ist in Abbildung 3.3 dargestellt. Der TIO-ASIC erlaubt den direkte Anschluss an das NUMAlink4 Netzwerk, unterstützt zwei PCI-X Busse, einen AGP-8X Bus und den Scalable System Port (SSP), welcher die Schnittstelle zum FPGA bildet.

Der FPGA kann mit einer Bandbreite von 16 GB/s auf seinen lokalen SRAM zugreifen, auf den Host und dessen Hauptspeicher mit (durch das NUMAlink4-Netzwerk begrenzten) 6,4 GB/s.

3.3 Core Services

Auf Seite der rekonfigurierbaren Hardware bilden die RASC Core Services die Schnittstelle zum Algorithmus. SGI stellt sie sowohl als Verilog-Quellcode, als auch als vorsynthetisierten IP-Core (fertige Schaltung in Form einer Netzliste) bereit. Verilog-Wrapper-Module werden als Verknüpfung zum ei-

[2] Quad Data Rate static RAM dual in-line memory modules: pro Taktzyklus können bis zu vier Datenwörter übertragen werden

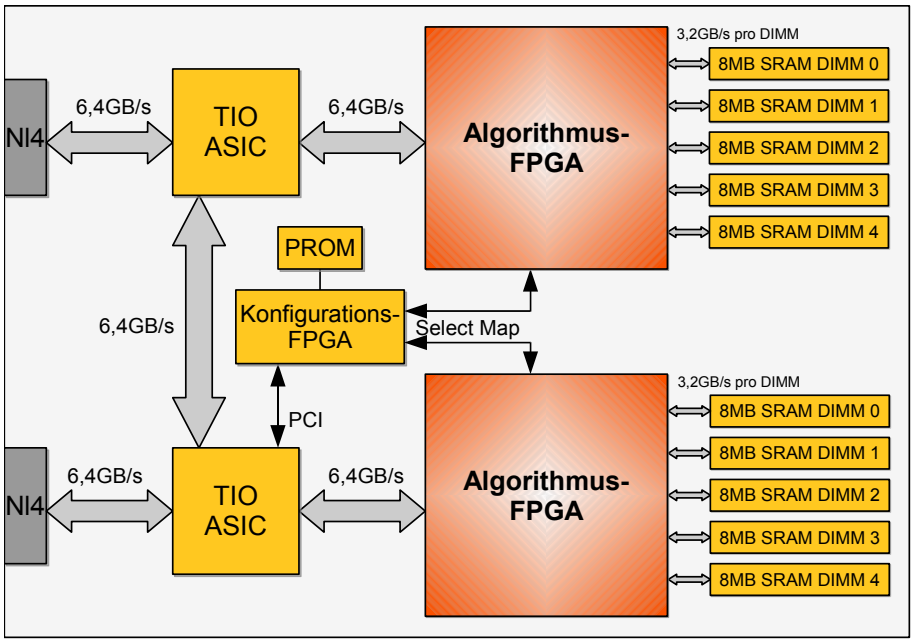


Abbildung 3.2: RC100-Blade (vgl. [SGI08])

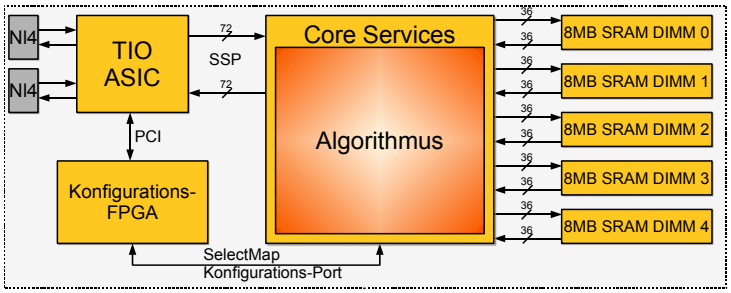


Abbildung 3.3: RASC FPGA-Modul (vgl. [SGI08])

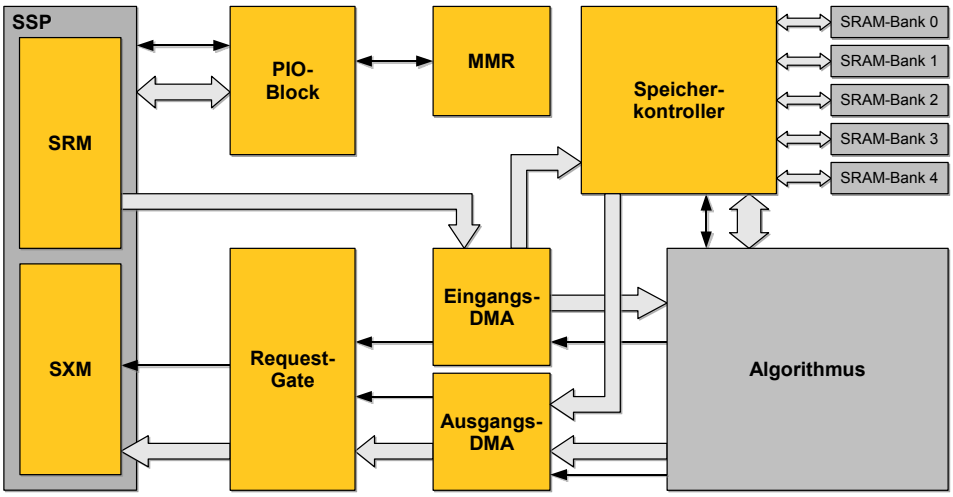


Abbildung 3.4: RASC Core Services (vgl. [SGI08])

gentlichen Algorithmus verwendet. Die Architektur der Core Services ist in Abbildung 3.4 dargestellt, wobei der Übersicht halber einzelne Komponenten des Kontrollpfades nicht enthalten sind, jedoch im Folgenden in ihrer Funktion noch erläutert werden. Die Erläuterung der Funktionalität einzelner Blöcke basiert inhaltlich auf [SGI08].

Das Empfangsmodul (SRM) und das Sendemodul (SXM) beinhalten entsprechende Logik um auf dem FPGA als Schittstelle zum *Scalable System Port* (SSP) zu fungieren. Es stehen drei beliebig kombinierbare Möglichkeiten zur Verfügung um Daten zwischen Host und Algorithmus auszutauschen:

- Einzelne Lese- oder Schreibanfragen von 64-Bit-Werten werden durch die *Programmed Input/Output (PIO) request engine* behandelt.
- Datenströme werden über die DMA-Blöcke verarbeitet und können direkt in den Algorithmus geleitet werden. Sowohl der DMA-Eingangsblock, als auch der DMA-Ausgangsblock bestehen aus jeweils bis zu vier unabhängigen DMA-Stream-Einheiten.
- Daten können über die DMA-Blöcke zwischen Host und lokalem SRAM transportiert werden. Der Speicher-Controller bedient dann das Interface zum on-board-SRAM und verknüpft es mit dem Algorithmus und den DMA-Blöcken.

Der *Memory Mapped Register (MMR)* Block stellt die Register bereit, welche dem Algorithmus-Designer zur Verfügung stehen sollen. Zu diesen Registern gehören die Debug-Register und die *Algorithm Defined Registers*. Beide Registertypen sind 64 Bit breit und können in einer maximalen Anzahl von je 64 Registern verwendet werden. Das *Request Gate* erstellt das SSP-Paket für Lese- und Schreibanfragen des FPGA zum Hauptspeicher. Der *Algorithm Block* beinhaltet den vom Benutzer geschriebenen anwendungsspezifischen Code, der den eigentlichen Algorithmus implementiert.

Nicht in der Abbildung dargestellt sind der *TNUM tracker*, welcher Transaktionsnummern innerhalb des FPGA handhabt und der *Interrupt Generator*, welcher Paketdaten erzeugt, um den Host im Falle eines direkten Speicherzugriffes (DMA) zu unterbrechen (z.B. durch das alg_done-Flag des Algorithmen-Blocks).

Die von den Core Services benötigte FPGA-Fläche lässt sich minimieren, indem nicht alle Komponenten genutzt werden. Es ist z.B. möglich den Memory-Controller wegzulassen, wenn der Algorithmus den SRAM nicht benötigt.

Üblicherweise werden FPGAs über eine UART oder eine PCI-Steckkarte angesteuert. Bei einer UART kann eine Datenrate von bis zu 115200 Bits pro Sekunde (14,4 kByte/s) erreicht werden. Im Vergleich dazu können mit PCI-Express-FPGA-Boards deutlich höhere Datenraten erreicht werden (PCIe 2.0 x32: 16 GByte/s).

Für das SGI RC100-Blade stehen drei Möglichkeiten zur Verfügung, Daten zwischen Host und FPGA zu übermitteln. Hierzu zählen eine Streaming Schnittstelle, eine Schnittstelle zum SRAM der RC100-Blades und spezielle 64-Bit-Register (*Memory Mapped Registers*). Die Bandbreite des Datentransfers zwischen den RC100-FPGAs und einer Host CPU ist durch das NUMAlink4-Netzwerk auf insgesamt 6,4 GB/s begrenzt.

3.3.1 SRAM-Schnittstelle

Die Speicheranbindung über den SRAM der RC100-Blades bietet die Möglichkeit Daten vom Host in den SRAM des jeweiligen FPGA zu schreiben und Daten aus dem SRAM des FPGA zu lesen. Ab der Version 2.20 bieten die SGI Core Services drei verschiedene Konfigurationsvarianten der fünf pro FPGA vorhandenen 8MB-SRAM-Bänke an:

(1) zwei 128-Bit-Ports (je zwei 64-Bit-Ports zu einem 128-Bit-Port zusammengefasst), ein 64-Bit-Port

(2) fünf voneinander unabhängige 64-Bit-Speicherports

(3) keine Ports zum SRAM

Ist es nicht möglich, die für die Ausführung des Algorithmus benötigten Daten im Block-RAM des FPGA zu halten, können diese in den SRAM ausgelagert werden und müssen nicht zwischen Host und FPGA transportiert werden. Während für die Host-FPGA Kommunikation eine Bandbreite von 6,4 GB/s zur Verfügung steht, ist der SRAM selber mit 16 GB/s an den FPGA angebunden. Die SRAM-Kommunikation kann ebenso für Streaming verwendet werden, wie das in Abschnitt 3.3.2 beschriebene direkte Streaming. Betrachtet man Abbildung 3.3 genauer, stellt man fest, dass die Speicheransteuerung mit vollen 200 MHz laufen muss, um die von SGI angegebenen maximalen Datenraten zu erreichen.

Hierzu wird davon ausgegangen, dass von einer 36 Bit breiten Leitung zu bzw. von einer SRAM-Bank nur 32 Bit verwendet werden (4 Bit für Datenintegrität). Bei einem QDR-II-SRAM können gleichzeitig zwei Wörter gelesen und geschrieben werden, wodurch man auf eine Datenrate von 1,6 GB/s je SRAM-Bank und Richtung kommt.

$$\textit{Durchsatz} = \textit{FPGA-Taktfrequenz} \cdot \textit{Wörter} \cdot \textit{Wortbreite} = 200\ \textit{MHz} \cdot 2 \cdot 32\ \textit{Bit} = 1{,}6\ \textit{GB/s}$$

Da sowohl der Algorithmus als auch die Core Services Zugriff auf den SRAM haben und parallel ausgeführt werden, muss zur Vermeidung einer Zugriffskollision eine Arbitrierung vorgenommen werden. Falls solche Zugriffskonflikte auftreten können, müssen sie auch in der Implementierung des Algorithmus behandelt werden. Werden die SRAM-Zugriffe so eingeplant, dass keine Zugriffkonflikte stattfinden können, muss keine Arbitrierungsbehandlung implementiert werden. Spezielle Direktiven und sog. Extractor-Anweisungen können dazu genutzt werden den Zugriff auf die SRAM Ports sowohl für die Core Services, als auch für den Algorithmus einzuschränken und damit Zugriffkonflikte zu vermeiden.

Werden SRAM-Ports so konfiguriert, dass nur der Algorithmus darauf zugreifen kann, minimiert das zusätzlich den Logik-Overhead der Core Services, da entsprechende Logik beim SRAM-Kontroller und den DMA Blöcken und auch deren Verdrahtung entfallen.

Die SRAM-Schnittstelle verfügt über zwei Verfahren zur Datenflusskontrolle (Handshaking-Verfahren): „Busy Signal“ und „Crediting Scheme“. Es ist dem Programmierer überlassen, welches er in seinem Entwurf verwendet. Eine Erklärung zu den Signalen, die je nach Verfahren benutzt werden, ist in [SGI08] beschrieben.

Um Datenmengen zu verarbeiten, die nicht in den SRAM des FPGA-Blades passen, bietet SGI eine Multibuffering-Lösung an. Hierzu werden Eingangs- und Ausgangsdaten in jeweils mindestens zwei Segmente pro SRAM-Bank unterteilt. Während der Host Daten in ein Segment des SRAM lädt, kann der FPGA-Algorithmus auf ein anderes Segment zugreifen. Die Umschaltung zwischen den Segmenten erfolgt über ein pro genutzter SRAM-Bank von den Core-Services zur Verfügung gestelltes Offset-Register, welches die oberen Bits der Speicheradresse bestimmt (sofern es vom Algorithmus-Designer genutzt wird).

3.3.2 Streaming Engines

Beim *Direct Streaming* werden Datenpakete in einem Datenstrom direkt an den Algorithmus-Block geschickt, ohne den Umweg über den SRAM der FPGAs zu nehmen. Dies soll zum einen die Latenzzeit des Datentransfers minimieren und verringert zum anderen den Logik-Overhead durch die Core Services, da

der Speicherkontroller nicht benötigt wird. Es stehen je vier DMA-Streaming-Engines für lesenden und schreibenden Zugriff zur Verfügung. Die Verwendung von mehreren Streaming-Engines hat allerdings keinen Einfluss auf die Bandbreite.

Um Daten aus einem Stream zu lesen, wird gewartet bis dieser bereit ist und anschließend eine Leseanfrage (*read enable*) gestellt. Der Schreibvorgang erfolgt über eine Schreibaufforderung, wenn der Stream bereit ist. Die genaue Definition und Verwendung der Signale der Streaming DMA Engine ist in [SGI08] beschrieben. Um zu erkennen, dass ein Input-Stream endet gibt es für den Algorithmus zwei Möglichkeiten: Parameter mit der Anzahl der zu lesenden Werte über ein *Algorithm Defined Register* übergeben oder das *Stream-In-Complete-Signal* verwenden, welches nach dem Senden des Streams über einen entsprechenden Funktionsaufruf des RASC-API gesendet werden kann.

Diese Kommunikationsvariante ermöglicht es den FPGA als Streaming-Koprozessor zu verwenden, ohne einen großen Overhead durch die Core Services zu benötigen.

3.3.3 Memory Mapped Registers

Für eine Kommunikation mit kleineren Datenpaketen stehen spezielle 64-Bit-Register zur Verfügung. Vorgesehen für die Fehlersuche stehen maximal 64 Debug-Register zur Verfügung. Diese können durch den Algorithmus beschrieben und vom Host gelesen werden. Die maximal 64 *Algorithm Defined Registers* (ADRs) sind flexibler einsetzbar und können sowohl vom Algorithmus, als auch vom Host gelesen und beschrieben werden.

Von SGI sind die ADRs primär zur Übergabe von Parametern vorgesehen, sie können aber auch für eine echte Ping-Pong-Kommunikation verwendet werden. Der Algorithmus erhählt für jeden vom Host vorgenommenen Schreib- oder Lesevorgang eines ADRs eine Signalisierung (*updated* oder *polled*), womit ihm der Zustand der Register bekannt ist. Auf der Seite der Host-Applikation fehlt eine entsprechende Synchronisation, welche eine Veränderung eines ADRs signalisiert und muss durch „Polling“ unter Verwendung eines Debug-Registers oder ADRs vom Nutzer selber ergänzt werden.

3.4 Software

Bisher wurde hauptsächlich auf die RASC-Hardware eingegangen. Um diese aus einer Hochsprache ansteuern zu können, ist jedoch noch Funktionalität auf weiteren Ebenen notwendig. Abbildung 3.5 zeigt eine Übersicht über die Abstraktionsebenen von RASC. Mit dem *Device Manager* wird die Organisation der Algorithmen vorgenommen, welche bei Bedarf in die vorhandenen FPGAs geladen wer-

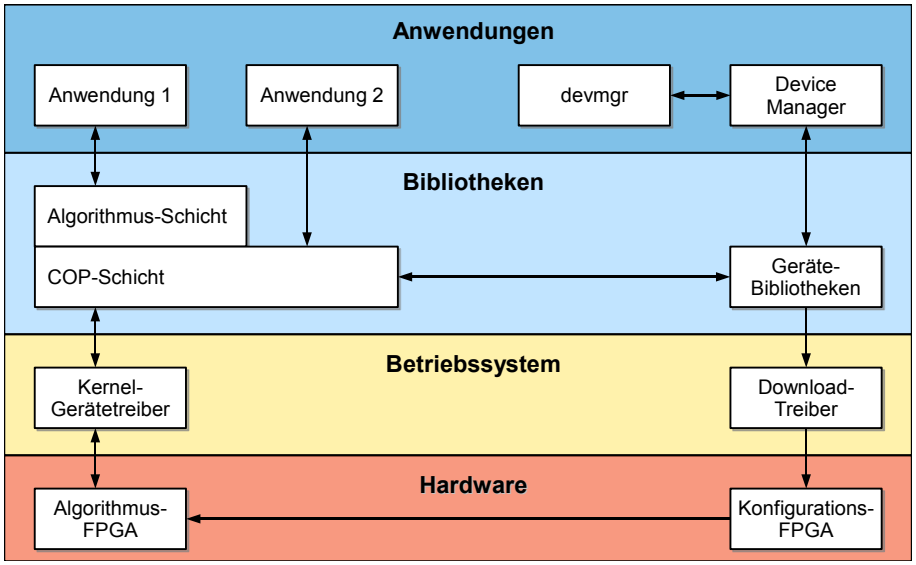


Abbildung 3.5: SGI RASC – Abstraktionsebenen (vgl. [SGI08])

den. Für den Anwendungsentwickler stehen neben Bibliotheksfunktionen zur Kommunikation mit den FPGAs auch hilfreiche Werkzeuge, wie ein erweiterter GNU Projekt Debugger und ein Algorithmus-Konfigurationstool zur Verfügung. Zudem werden einfache Beispielanwendungen von SGI angeboten, um den Einstieg zu erleichtern.

3.4.1 Abstraction Layer (API)

Der RASC Abstraction Layer (RASCAL) ist eine Programmierschnittstelle (API) für die „kernel device driver“ und die RASC-Hardware und bietet damit eine Schnittstelle zur Ansteuerung der RC100-FPGAs aus einem C- oder Fortran-Programm heraus.

RASCAL besteht aus zwei Ebenen, wobei nur eine von beiden innerhalb eines Programmes verwendet werden darf: die Co-Prozessor(COP)-Ebene und die Algorithmen-Ebene. Die zugrundeliegende COP-Ebene stellt Funktionen zur individuellen Ansteuerung der FPGAs als COPs bereit. Darauf aufsetzend ermöglicht die Algorithmen-Ebene mehrere COPs als eine logische Implementierung eines Algorithmus anzusprechen und übernimmt damit die Aufteilung von Daten auf die verschiedenen FPGAs und auch die Steuerung des Multibuffering (siehe Abschnitt 3.3.1) auf Seite des Hosts.

Tabelle 3.2 gibt einen Überblick über einige *rasclib*[3]-Funktionen, wobei die aufgeführten Funktionen der COP-Ebene entsprechende Äquivalente in der Algorithmen-Ebene besitzen. Die in der Tabelle stehenden Funktionsaufrufe sind in der Reihenfolge aufgelistet, wie sie üblicherweise benutzt werden. Optional zu verwenden sind die Sende- und Empfangs- bzw. die Lese- und Schreibfunktionen, wobei die Ausfüh-

[3]Bibliothek des RASC Abstraction Layer, welche Funktionen zur Ansteuerung der FPGA-Koprozessoren bereitstellt

rung eines Algorithmus ohne Übertragung von Daten natürlich sinnfrei wäre. Sende- und Empfangs-Funktionsaufrufe werden pinzipiell in eine Warteliste eingereiht und erst durch die Commit-Funktion an die Kernel-Device-Treiber geschickt, während die Lese- und Schreibfunktionen, je nach übergebenem Flag, auch ohne Commit sofort ausgeführt werden können. Sende- und Empfangsfunktionen werden für das Übertragen größerer Datenmengen genutzt (siehe Abschnitt 3.3.2 und 3.3.1) und können mit *Direct I/O* oder *Buffered I/O* arbeiten. Für die Verwendung der Lese- und Schreibfunktionen zum Zugriff auf die FPGA-Register siehe 3.3.3.

Funktion	Beschreibung
`rasclib_resource_reserve`	reserviert eine bestimmte Anzahl von FPGAs
`rasclib_resource_configure`	programmiert die reservierten FPGAs mit dem angegebenen Bitstream
`rasclib_cop_open`	signalisiert der rasclib, dass ein FPGA mit angegebenem Algorithmus genutzt werden soll und gibt den Deskriptor des verwendeten FPGAs zurück
`rasclib_cop_reg_write`	schickt einen Wert an ein angegebenes FPGA-Register
`rasclib_cop_send`	sendet Daten an den FPGA (Warteliste)
`rasclib_cop_go`	startet den FPGA-Algorithmus
`rasclib_cop_receive`	empfängt Daten vom FPGA-Ausgangs-Puffer (Warteliste)
`rasclib_cop_reg_read`	liest einen Wert aus einem angegebenen FPGA-Register
`rasclib_cop_commit`	sendet alle in der Warteliste eingereihten Befehle
`rasclib_cop_wait`	blockiert die Programmausführung bis alle gesendeten Befehle ausgeführt wurden
`rasclib_cop_close`	gibt alle Host-Ressourcen im Zusammenhang mit dem angebenen Algorithmus frei
`rasclib_resource_return`	gibt die Konfiguration eines FPGA frei (bleibt reserviert)
`rasclib_resource_release`	gibt reservierte FPGAs frei

Tabelle 3.2: Ausgewählte Funktionen der *rasclib*-Bibliothek

Mit *Direct I/O* können Daten direkt aus den vom Nutzer angelegten Puffern zum FPGA-Koprozessor geschickt werden, ohne dazwischen in einen Speicherbereich des Kernels kopiert werden zu müssen. Der Speicherbereich muss hierbei an 128-Byte-Grenzen ausgerichtet und möglichst kontinuierlich auf den physikalischen Speicher abgebildet sein. Unter Nutzung der hugetlbfs-Funktion des Linux-Kernels kann die *rasclib* passende Speicherbereiche bereitstellen. Bei Buffered-IO hingegen wird der Speicherinhalt vorher in einen Speicherbereich des Kernels kopiert. Dies ist zwar an keine Voraussetzungen geknüpft, bietet aber weniger effektive Bandbreite ([SGI08] Kapitel 5).

3.4.2 Algorithmen-Verwaltung

Um einen Algorithmus auf den RC100-FPGAs ausführen zu können, muss dieser zuvor als binäre Datei (engl. Binary) unter einem eindeutigen Namen (ID) im System registriert werden. Die Erstellung des Al-

gorithmus wird in Kapitel 4 genauer erläutert. Nachdem das Binary in der Registry eingetragen wurde, kann es über den RASC Abstraction Layer (siehe 3.4.1) mit seinem zugehörigen Identifikator angesprochen werden. Zu jedem auf den RC100-Blades ausführbaren Algorithmus gehören ein Binary und zwei Konfigurationsdateien für die Core Services und die davon verwendeten Schnittstellen.

Der *Device Manager* übernimmt die Verwaltung der Algorithmen, die den FPGAs zur Verfügung stehen. Er ermöglicht das Anzeigen der im System vorhandenen FPGAs, das Auflisten der zur Verfügung stehenden Algorithmen und das Hinzufügen, Ändern und Entfernen von Algorithmen. Als „Frontend“ wird das Kommandozeilenprogramm *devmgr* verwendet.

3.4.3 GNU Project Debugger (GDB)

Eine Möglichkeit der Fehlersuche zur Laufzeit einer RASC-Anwendung bietet SGI mit der Erweiterung des GDB an. Folgende RASC-spezifische Kommandos stehen zusätzlich zur Verfügung:

- fpgaactive [on/off]
- set fpga fpganum = N
- info fpgaregisters [regname] (alias: ’info fr’)
- info fpga
- fpgastep
- fpgacont
- fpgatrace [on/off]

Mit diesen Kommandos ist es möglich, in jedem Takt den Inhalt der Debug- und Algorithm-Defined-Register auszulesen und sich generelle Informationen, zum auf dem FPGA laufenden Algorithmus, anzeigen zu lassen. Eine genauere Beschreibung der Kommandos kann in [SGI08] nachgelesen werden.

Eine der wesentlichen Beschränkungen die das Debugging über den GDB mit sich bringt, betrifft das Auslesen der FPGA-Register. Nur vom Hardware-Programmierer nach außen propagierte Register können ausgelesen werden. Damit muss schon während des Hardware-Entwurfs darauf geachtet werden, entsprechende Register mit Debugging-Informationen zu belegen, um später, falls die FPGA-Anwendung nicht die gewünschte Funktionalität aufweist, eine Fehlersuche mit dem GDB zu ermöglichen. Zudem wird derzeit das direkte Streaming über die Streaming Engines mit dem GDB FPGA nicht unterstützt.

Für den Debugging-Modus sollte am sog. *step_flag_out*-Register, welches der Hardware-Designer setzen kann, eine ’1’ anliegen. Dieses signalisiert den Core Services (siehe Abschnitt 3.3), dass ein Taktzyklus beendet wurde. Liegt andernfalls eine ’0’ an, kann kein taktgenaues Debugging vorgenommen werden.

3.4.4 Algorithmus-Konfigurations-Tool

Mit der RASC Version 2.20 wird die Konfiguration des Algorithmus benutzerfreundlicher. Um die Core Services und den Algorithmus auf die benutzten Ressourcen abzustimmen, werden Konfigurationsdateien und Wrapper-Module benötigt. Bei vorherigen RASC-Versionen mussten diese Dateien entsprechend vorhandener generischer Vorlagen vom Hardware-Entwerfer manuell erzeugt werden.

Das *SGI RASC Algorithm Configuration Wizard* ist ein Tool Command Language (TCL) Skript und ermöglicht es anhand einer grafischen Oberfläche (GUI) die Ressourcen auszuwählen, die dem Algorithmus zur Verfügung stehen sollen. Folgende Einstellungen können vorgenommen werden:

- Bezeichnung des Algorithmus
- Taktfrequenz des Algorithmus (50MHz, 66MHz, 100MHz, 200MHz)
- zu verwendende DMA Streams (siehe 3.3.2)
- SRAM-Konfiguration(siehe 3.3.1)
- Anzahl und Konfiguration der Algorithm Defined Register (siehe 3.3.3)
- Anzahl der Debug-Register
- Auswahl des Synthese-Tools
- Multiplikator und Teiler der *Supplemental Algorithm Clock*[4]

Entsprechend der Auswahl erstellt das Tool Konfigurationsdateien, Verilog-Wrapper-Module um das Design des Algorithmus in die Core Services zu integrieren und ein Makefile um den Bitstream automatisiert erzeugen zu können. Der Name des Algorithmus wird zur Benennung der Verzeichnisse und Design-Dateien verwendet.

[4] aus der Taktfrequenz des Algorithmus abgeleitete Taktfrequenz

4 Programmiermöglichkeiten der SGI RASC Plattform

In diesem Kapitel werden zwei Möglichkeiten zur Programmierung der SGI RASC Plattform vorgestellt. Dies ist zum einen die Mitrion Entwicklungsumgebung (Mitrion SDK) mit Mitrion-C als parallele Hochsprache und zum anderen die Hardwarebeschreibungssprache VHDL als eine herkömmliche Methode des Hardware-Entwurfs. Die Alternative zu VHDL ist Verilog, in der auch Teile der RASC Core Services implementiert sind. Die Entscheidung für die Verwendung von VHDL und entsprechende Gründe sind in Abschnitt 4.2 genauer erläutert. Desweiteren lag das größte Fallbeispiel, das Damenproblem (siehe Abschnitt 5.5), bereits in VHDL-Code vor und ich beherrsche die Programmierung mit VHDL besser als mit Verilog.

Die Software-Entwicklung soll in beiden Fällen (Mitrion-C, VHDL) als Vergleich dienen, wobei auf wesentliche Unterschiede zwischen beiden Programmiermöglichkeiten direkt eingegangen wird. Zu Beginn des Kapitels werden noch begrenzende Faktoren für die Parallelität eines Systems aufgezeigt, zumal performante Problemlösungen mit SGI RASC auf einen hohen Grad an Parallelverarbeitung angewiesen sind.

Anforderungen der Parallelverarbeitung

(1) Verarbeitungseinheiten

Ein begrenzender Faktor für die Parallelität eines Systems ist die Anzahl der verfügbaren Verarbeitungseinheiten. Jede dieser Verarbeitungseinheiten benötigt, entsprechend ihrer Aufgabe eine gewisse Fläche auf dem Chip. Einfache Operationen benötigen weniger Fläche und können damit in größerer Anzahl auf den Chip gebracht werden, wodurch sich die mögliche Parallelverarbeitung erhöht.

(2) Datenabhängigkeiten

Ein weiterer Faktor, der die Parallelität eines Systems begrenzt, ist die Anzahl der unabhängig voneinander verarbeitbaren Daten. Zwei Daten sind genau dann unabhängig voneinander, wenn weder das eine, noch das andere Datum auf die Berechnung des zweiten Datums einen Einfluss hat. Ist eine Abhängigkeit zwischen zwei Daten vorhanden, so können sie nur sequentiell abgearbeitet

werden. Viele Datenabhängigkeiten führen zu einer Verminderung der Parallelverarbeitung und damit zu einer Verlängerung der Laufzeit.

(3) Simultan zugreifbarer Speicher

Die Anzahl der Speicherplätze, auf welche gleichzeitig zugegriffen werden kann, ist ebenso ein begrenzender Faktor hinsichtlich der Parallelität eines Systems. Um also mehrere Werte gleichzeitig berechnen zu können, muss auch eine ebenso große Anzahl zur gleichen Zeit beschreibbarer Speicherplätze vorhanden sein. Damit muss jede Verarbeitungseinheit, wenn alle gleichzeitig arbeiten sollen, die Möglichkeit haben ihre erzeugten Werte zu puffern.

Diese trivialen Regeln (vgl. [Mit08b], Kapitel 12) sollten beachtet werden, wenn die mögliche Parallelität eines Algorithmus eingeschätzt und eine entsprechenden Implementierung davon erstellt werden soll. Insbesondere sind die Datenabhängigkeiten bei der Entwicklung und Bewertung paralleler Algorithmen von Bedeutung. Während sie durch den Mitrion-C-Compiler automatisch erkannt werden, muss sich der Programmierer einer Hardwarebeschreibungssprache explizit um deren Erfüllung kümmern. Mitrion-C erleichtert durch implizit parallele Programmierung und die Ausführung der Software auf dem MVP (siehe 4.1.2) die angegebenen Anforderungen zu erfüllen ((1) und (3) werden größtenteils automatisch von Mitrion gehandhabt), im Algorithmus vorhandene Datenabhängigkeiten (2) führen aber dennoch zu sequentieller Abarbeitung. Es ist also für beide, die Programmierung von Mitrion-C und VHDL, ein effizienter paralleler Algorithmus notwendig, um die Vielzahl an Verarbeitungseinheiten auf FPGAs ausnutzen zu können. Trotz dieser hardwarenahen Anforderungen ist im Vergleich zu Hardwarebeschreibungssprachen bei der Programmierung von Mitrion-C kaum Wissen über die zugrundeliegende Hardware notwendig.

Auch das Verbindungsnetzwerk hat in großen Rechensystemen starken Einfluss auf die Performance von Algorithmen. Wenn zu dessen Ausführung Daten über das Netzwerk übertragen werden müssen, können die Bandbreite und die Latenz begrenzende Faktoren sein, sofern der Algorithmus entsprechende Anforderungen an diese besitzt.

4.1 Mitrion SDK von Mitrionics

Das Mitrion Software Development Kit (SDK) von Mitrionics ist eine Entwicklungsumgebung, welche um die parallele Programmiersprache Mitrion-C aufgebaut ist und dient der Umsetzung hochparalleler Algorithmen auf programmierbaren Hardware-Plattformen mit FPGAs. Derzeit werden sechs hybride Rechnersysteme[1] von der Mitrion-Software-Plattform unterstützt. Neben Mitrion-C Bibliotheken bein-

[1] Hardware-Plattformen mit Standardprozessoren und Hardware-Beschleunigern

haltet das SDK noch die „Mitrion Virtual Processor“ (MVP) Architektur, einen Mitrion-C-Compiler und einen Simulator. Die Anwendungsbereiche befinden sich in der Bioinformatik, der Gen-Sequenzierung, der Textsuche und der Bildverarbeitung. Unabhängig davon können auch beliebige parallele Algorithmen implementiert werden. Das Mitrion Benutzerhandbuch [Mit08b] stellt die Mitrion Plattform vor. Es beschreibt neben der Mitrion SDK und auch den Funktionsumfang von Mitrion-C und bietet damit eine Einführung in die Programmierung dieser Sprache.

Ein in Mitrion-C geschriebenes Programm kann auf entsprechend unterstützten Plattformen in Hardware umgesetzt werden, genauere Details der zugrundeliegenden Hardware werden dem Programmierer jedoch verborgen. Der Syntax ist dem von ANSI-C sehr ähnlich und soll die Einarbeitungszeit für den Entwickler verkürzen und die Portierung von C-Code zu Mitrion-C-Code erleichtern.

Das Prinzip der Hardwarebeschreibung basiert auf dem MVP, welcher in Mitrion-C programmierte Software auf dem FPGA ausführt und damit Software von Hardware isoliert. Die Anpassung eines Programmes an den MVP wird durch die *Processor Configuration Unit* der Mitrion SDK vorgenommen. Dieser Vorgang besteht aus der Entscheidung über die benötigten verarbeitenden Elemente (PEs) und der Verschaltung dieser. Das Prozessor-Design wird vom Mitrion SDK als VHDL-Code ausgegeben und kann anschließend synthetisiert, auf die Zieltechnologie abgebildet, plaziert und verdrahtet werden. Mitrion-C und der MVP erben die Beschränkungen, die die Programmierung von FPGAs mit sich bringt:

- Durch die Umsetzung der Befehle in Logik und damit FPGA-Ressourcen, ist die maximale Länge von Programmen beschränkt.
- Die Speicherverwaltung ist vergleichsweise rudimentär. Der Nutzer muss dafür sorgen, dass Daten in verschiedenen Adressräumen zur Verfügung stehen. (FPGAs haben im Vergleich zu Mikroprozessoren mehrere Adressräume, deren Größe und Typ schon während der Programmübersetzung bekannt sind.) Außerdem besitzen FPGAs keine Caches, welche auf Grund der großen Anzahl frei definierbarer Register und der vorhandenen Block-RAM-Ressourcen (ein Takt Zugriffslatenz) auch nicht nötig sind.
- Es gibt kein Statusregister oder ähnliches und damit keine exakten Fehler wie „Division durch Null“ oder „Index außerhalb des Zahlenbereiches“.
- Die Fehlersuche auf FPGAs ist komplexer und es ist nicht möglich jede Variable zu jedem Zeitpunkt auszulesen. Zudem gibt es wenige Debugger zur Fehlersuche in FPGA-Programmen.

Die Anwendungsentwicklung mit der Mitrion Plattform ist einem typischen Software Entwicklungszyklus ähnlich. Bei der Portierung auf ein anderes rekonfigurierbares Computersystem braucht der Nutzer nur Detailles bezüglich der Organisation der Maschine anzupassen. Der Mitrion-C-Quellcode bleibt unabhängig von der Größe, der Geschwindigkeit und dem verfügbaren Speicher des FPGAs unverändert

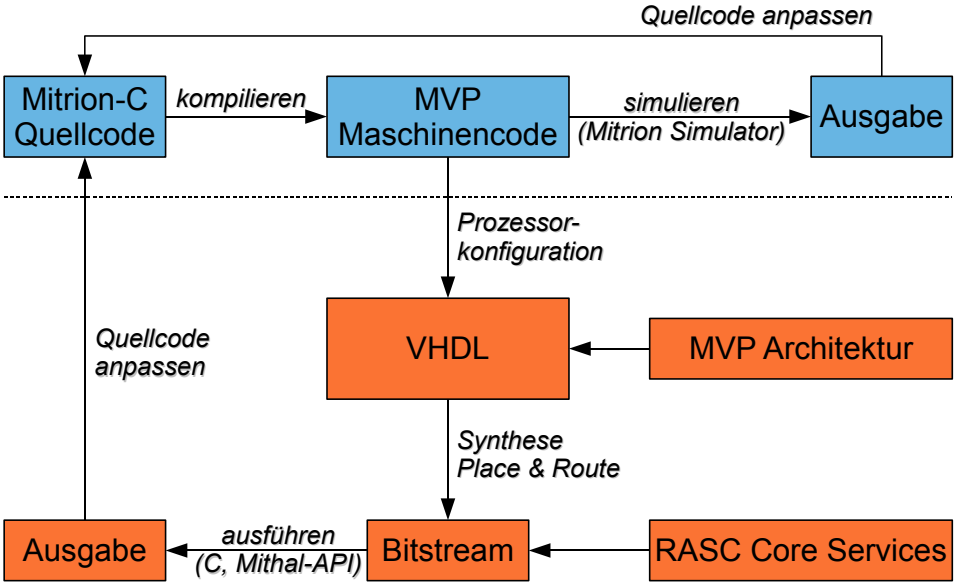


Abbildung 4.1: Entwicklungszyklus – Mitrion SDK auf SGI RASC (vgl. [Mit08b])

und ist nach erneuter Umsetzung in Hardware wieder im vorhandenen Umfang nutzbar. Wird auf ein kleineres System (FPGAs mit weniger Logik) portiert, kann der Algorithmus u.U. nicht mehr implementiert werden oder es muss die Parametrisierung bei einem generischen Entwurf verändert werden. Abbildung 4.1 zeigt den Entwicklungszyklus für Mitrion-C-Programme auf der SGI RASC Hardware. Im oberen Bereich ist die Softwareentwicklung, im unteren Bereich die Umsetzung in Hardware dargestellt.

4.1.1 Programmiersprache Mitrion-C

Um den MVP effektiv programmieren zu können, stellt Mitrionics mit Mitrion-C eine implizit parallele Programmiersprache bereit. Diese soll den Programmierer dabei unterstützen, die Anforderungen einer parallelen Ausführung eines Programmes zu erfüllen und zudem leicht und schnell erlernbar sein. Der Syntax lehnt sich an bekannte Sprachen, wie C, C++, Java, C#, etc. an. Zusätzlich soll Mitrion-C verhindern Verklemmungen (Deadlocks) oder Wettlaufsituationen (Race Conditions) zu beschreiben. Verwendet man sog. Instance Tokens und Streams, ist dies aber dennoch möglich.

In Mitrion-C kann der Programmierer eine beliebige Anzahl von untereinander abhängigen Prozessen beschreiben, wobei die Synchronisation zwischen diesen dem Programmierer verborgen wird. Das Scheduling-System des MVP sorgt dafür, dass Daten zur richtigen Zeit am richtigen Ort sind. Prozesse werden implizit durch einzelne Anweisungen oder Anweisungsblöcke erzeugt und innerhalb von Funktionen explizit vom Programmierer gebündelt. Die Zusammenfassung von Anweisungen in Funktionen ist nicht performance-relevant und dient lediglich der Strukturierung. In Hardwarebeschreibungsspra-

```
Mitrion-C 1.5;
// options: -cpp
#define ExtRAM  mem uint:128[2048]

(ExtRAM, ExtRAM) main(ExtRAM sram0, ExtRAM sram1){

    // durch Bandbreite begrenzte Schleife
    results = foreach(i in <0 .. 4>) {
        int:16 partsum = i;
        // durch Latenz begrenzte Schleife
        sum = for(j in <1 .. 10>){
            partsum = partsum + j;
        } partsum;
        watch sum;
    } sum;

    sram1_final = foreach(res in results by i) {
        sram1w= memwrite(sram1,i,res);
    } sram1w;

}(sram0,sram1_final);
```

Auflistung 4.1: Mitrion-C Beispielprogramm

chen (vgl. 4.2.1) hingegen muss die Synchronisation von Prozessen und das korrekte, taktgenaue Bereitstellen von Daten der Programmierer vornehmen. Prozesse und nebenläufige Anweisung werden in VHDL explizit vom Programmierer beschrieben.

Obwohl der Syntax von Mitrion-C sich an bekannte Hochsprachen der C-Familie anlehnt, gibt es einige wesentliche Konzepte, welche von der ANSI-C-Programmierung sehr abweichen. Während in Mitrion-C Parallelisierung und Datenabhängigkeiten im Mittelpunkt stehen, ist dies bei traditionellen sequentiellen Programmiersprachen die Reihenfolge der Befehlsausführung. In Mitrion-C gibt es keine Ausführungsreihenfolge im herkömmlichen Sinne. Es wird jede Operation, sobald ihre Datenabhängigkeiten erfüllt sind, sofort ausgeführt. Parallelität ist in Mitrion-C-Programmen implizit.

Ein weiterer Untschied zwischen Mitrion-C und anderen Hochsprachen der C-Familie ist, dass die meisten Anweisungen (außer Deklarationen und „Watch-Statements“) in Mitrion-C Zuweisungen sind. Damit sind `if`-, `for`- und `while`-Anweisungen in Mitrion-C Ausdrücke, die einen Ergebniswert zurückgeben. Die `Ternary(?:)`-Operation (bedingte Zuweisung) aus C ist ein Beispiel für eine `if`-Anweisung in Form einer Zuweisung. In Mitrion-C muss eine Anweisung innerhalb eines Blockes über dessen Rückgabewert propagiert werden, um außerhalb des Blockes Einfluss nehmen zu können. Ein kurzes Mitrion-C-Programm verdeutlicht der Quellcode 4.1.

Mitrion-C gehört zu den *Single-Assignment Languages*. Das bedeutet, dass in einem Block einer Variable nur ein einziges Mal ein Wert zugewiesen werden darf. Diese wichtige Einschränkung ermöglicht die

parallele Ausführung der Anweisungen eines Blockes (sofern keine Datenabhängigkeiten bestehen).

Für die Fehlersuche auf Kommandozeile stellt Mitrion-C die *watch*-Anweisung zur Verfügung. Wie im Code-Ausschnitt für `part_sum` verwendet, kann damit der Wert einer Variablen während der Programmausführung beobachtet werden.

Alle Mitrion-C-Quellcode-Dateien müssen mit Angabe der Sprachversion (*language version specifier*) beginnen. Diese bestimmt die Regeln des Parsers, die Semantik einiger integrierter Funktionen, Datentypen und die Verfügbarkeit von syntaktischen Konstrukten. Der Mitrion-C-Compiler unterstützt alle vorherigen Versionen, solange die Sprachversion im Quelltext angegeben wird.

Ein Mitrion-C-Programm wird durch die Abarbeitung der *main()*-Funktion ausgeführt. Ihr werden sog. *Instance Token* für die verwendeten Speicherbänke übergeben. Als Rückgabewert liefert sie die finalen *Instance Token*, nachdem alle Zugriffe durchgeführt wurden.

Um Makros zu benutzen ruft der Mitrion-C-Compiler den C-Preprozessor auf. *Include*-Direktiven dürfen nur außerhalb von Funktionen stehen. Die Verwendung von *include* für externe nicht Mitrion-C-Dateien ist für die gemeinsame Verwendung von Makros in Mitrion-C- und Host-Programm sinnvoll.

Mitrion-C arbeitet mit einem eigenem API, welches auf RASCAL (siehe 3.4.1) aufbaut, um einige Funktionen erleichtert und an anderer Stelle erweitert wurde. Abbildung 4.2 zeigt überblicksweise wie sich Mitrion in SGI RASC integriert.

Mitrion-C ist keine Hardwarebeschreibungssprache, sondern eine implizit parallele Programmierspra-

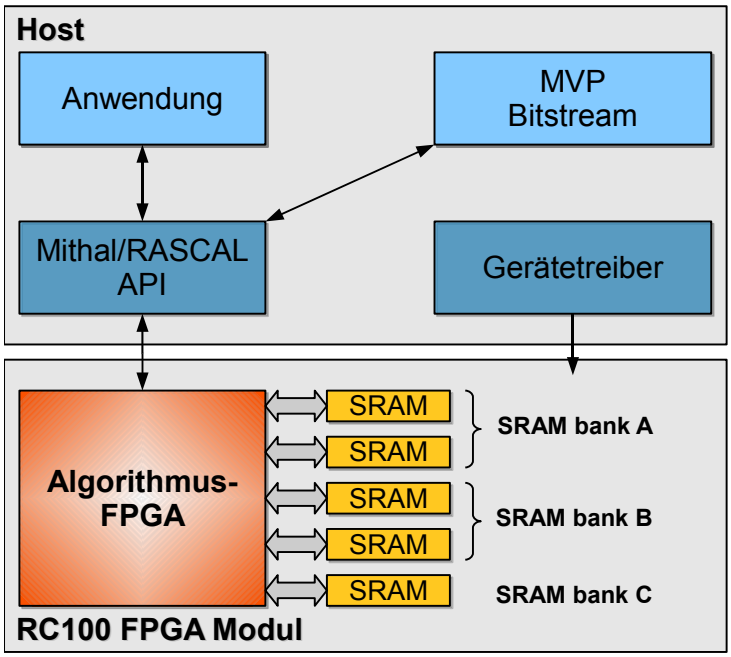


Abbildung 4.2: Integration von Mitrion in SGI RASC (vgl. [Mit08a])

che, die dem Entwickler die Möglichkeit geben soll, die Parallelität der Verarbeitungsblöcke des FPGA zu nutzen und ihn als Koprozessor in Verbindung mit einem Mikroprozessor zu betreiben. Eine Schaltung ansich zu entwerfen, ist mit Mitrion-C nicht möglich. Allerdings bietet Mitrion die Möglichkeit, VHDL-IP-Block-Plugins über externe Funktionsaufrufe innerhalb eines Mitrion-C-Programms zu nutzen. Damit lassen sich Teile des Algorithmus, die von Mitrion suboptimal umgesetzt werden, per Hand in VHDL optimieren. Allerdings muss dann auch beachtet werden, dass der MVP mit 100MHz läuft und unter dieser Anforderung der VHDL-Entwurf vorgenommen werden.

4.1.2 Mitrion Virtual Processor

Der Mitrion Virtual Processor (MVP) ist ein feingranularer, massiv paralleler, rekonfigurierbarer Prozessor, welcher auf FPGAs implementiert wird. Er liegt als Architektur vor und wird durch einen in Mitrion-C beschriebenen Algorithmus konfiguriert und schließlich als Soft-IP-Core ausgegeben. Im MVP wird jede Operation oder Menge von Operationen einer Verarbeitungseinheit, einem sog. „Processing Element“ (PE), zugeordnet. Da ein PE nur einen sehr kleinen Teil des Programmes ausführt, normalerweise nur eine oder wenige Operationen, entsteht eine sehr feingranulare Prozessorstruktur. Massive Parallelverarbeitung entsteht dadurch, dass eine Vielzahl von PEs gleichzeitig arbeiten und interagieren können. Für jedes Mitrion-C-Programm wird der MVP individuell konfiguriert und beinhaltet nur diejenigen PEs, die das Programm auch benötigt.

Die zwei wesentlichen Merkmale von FPGAs im Vergleich zu Mikroprozessoren sind die niedrige Taktfrequenz von nur wenigen hundert Megahertz (MHz) und die große Anzahl von Logikelementen, welche parallel arbeiten können. Damit muss der MVP die Möglichkeit der massiven Parallelverarbeitung nutzen, um die Ressourcenausnutzung des FPGAs zu maximieren. Prozessoren mit einer von-Neumann-Architektur sind bereits als IP-Cores für FPGAs vorhanden (MIPS-, PowerPC-IP-Cores), leiden aber unter der geringen Taktfrequenz der FPGAs und können durch ihren sequentiellen Datenstrom auch nicht von der Vielzahl von parallel arbeitenden Bausteinen auf dem FPGA profitieren. Dementsprechend muss der MVP folgenden Anforderungen genügen, um eine effiziente Abarbeitung zu gewährleisten:

- Parallelität des Befehlssatzes: Befehle können gleichzeitig ausgeführt werden
- Parallelität auf Schleifenebene: mehrere Schleifendurchläufe können parallel ausgeführt werden
- Maximale Nutzung der FPGA Ressourcen durch Anpassung des Prozessors an den auszuführenden Algorithmus

Wenn beispielsweise die Bedingung einer if-Anweisung erst zur Laufzeit ausgewertet werden kann, muss der MVP beide Programmzweige in Form von PEs implementieren. Dadurch ist es sinnvoll bereits wäh-

rend der Auswertung der Bedingung beide Zweige gleichzeitig auszuführen und dann das Ergebnis des entsprechenden Zweiges zurückzugeben. Wird innerhalb eines Zweiges auf Speicher zugegriffen, muss zuerst die Bedingung ausgewertet werden bevor ein Zweig abgearbeitet werden kann.

Der MVP arbeitet nicht wie ein üblicher Mikroprozessor mit einem Befehlsstrom. Anstelle dessen werden die zu verarbeitenden Befehle auf dem FPGA in Form von Logik implementiert und ein Datenstrom durch diese geleitet. Es ist nicht möglich den MVP in eine bestehende Schaltung einzubinden, da dieser nur in Verbindung mit der entsprechenden Anbindung unterstützter rekonfigurierbarer Systeme und einem angepassten API funktioniert.

4.1.3 Mitrion Simulator

Mitrionics stellt mit dem Mitrion Simulator ein Werkzeug zur Verfügung, mit dem Simulation der Funktionalität und Fehlersuche in Mitrion-C-Programmen vorgenommen werden kann, ohne dass der FPGA dafür benötigt wird. Folgende zwei Punkte verdeutlichen die Notwendigkeit der Simulation:

- Die Debugging-Möglichkeiten von auf dem FPGA laufenden Anwendungen sind stark begrenzt (siehe 3.4.3).
- Synthese, Plazierung und Verdrahtung benötigt viel Zeit und verlangsamen die Fehlersuche.

Der Simulator kann in drei Modi arbeiten, die im Folgenden näher erläutert werden. Nachdem der Mitrion-C-Compiler das Programm ohne Fehler übersetzt hat, werden entsprechend dem verwendeten Modus Debug-Ausgaben angezeigt.

Simulation mit grafischer Benutzeroberfläche

Der „Graphical User Interface“ (GUI) Modus erzeugt einen Datenabhängigkeitsgraphen, der die Datenparallelität und die Parallelität durch Pipelines visualisiert. Jeder Knoten des Graphen repräsentiert eine vom Programm durchgeführte Operation. Jede Kante, welche zwei Knoten verbindet, stellt eine gerichtete Datenabhängigkeit dar. Normalerweise ist ein Knoten von darüberliegenden Vorgängerknoten abhängig. Umgekehrte Abhängigkeiten, wenn also ein Knoten von seinem darunter liegendem Vorgänger abhängt, werden mit gelben Kanten dargestellt. Knoten mit einem schwarzen Rand sind Mitrion-C-Funktionen oder Schleifen und beinhalten einen Untergraphen.

Wird der Simulator ausgeführt, fließen Daten über die Kanten durch die Knoten, welche entsprechende Operationen auf diesen ausführen. Verschiedene Farben weisen auf den Zustand eines Knotens während der Simulation hin:

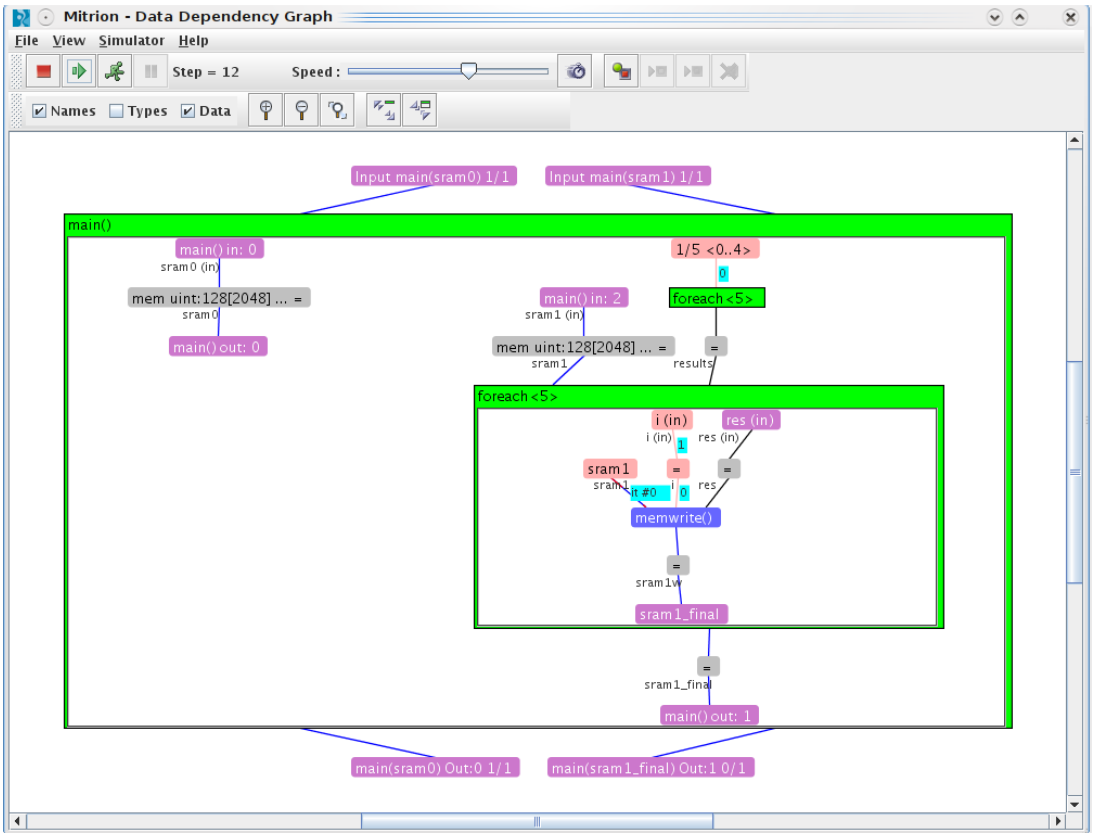


Abbildung 4.3: Mitrion-C Simulation mit GUI

grün: Der Knoten hat im letzten Zeitschritt seine Operation ausgeführt.

rot: Der Knoten ist angehalten und kann seine Operation nicht ausführen, weil der Ausgang nicht konsumiert werden konnte.

grau: Der Knoten wartet auf Eingangsdaten.

Daraus kann der Programmierer erkennen, wie gut die parallele Abarbeitung des Algorithmus funktioniert. In einem gut parallelisierten Programm ist die überwiegende Anzahl der Knoten grün. Abbildung 4.3 zeigt den graphischen Simulator für das Mitrion-C-Programm aus 4.1.

Blaue Knoten und Kanten arbeiten mit Instance Tokens (also RAM-Zugriffen), während violette Knoten Ein- oder Ausgänge von Untergraphen darstellen. Kanten, die während der Simulation einen Wert übermitteln, zeigen diesen hellblau hinterlegt an. In der Abbilung wurde der Knoten (foreach-Schleife) für das Schreiben der Ergebnisse in den SRAM bereits aufgeklappt und wartet auf die Ergebnisse der vorherigen Schleife. Eine vollständige Darstellung aller Formen und Farben von Knoten und Kanten ist in [Mit08b] zu finden.

Der Datenabhängigkeitsgraph, welcher durch den Simulator im interaktiven Modus genutzt wird, benötigt noch einige Optimierungsschritte bevor daraus ein MVP erzeugt werden kann. Dementsprechend kann der Graph nur genutzt werden, um die Funktionalität des Mitrion-C-Programmes zu simulieren.

Die wirkliche Performance des Programmes kann nur durch die Ausführung des Simulators im Batch-Modus abgeschätzt werden, in welchem die Simulation auf dem optimierten Graphen durchgeführt wird. Für zukünfige Mitrion-Versionen hat Mitrionics einen taktgenauen Graphen (vgl. Timing-Simulation in Abschnitt 4.2.2) angekündigt, der allerdings erst nach der Hardware-Synthese (siehe Abschnitt 4.2.3) angezeigt werden kann.

Es werden einige Möglichkeiten geboten um durch den Graphen zu navigieren. Neben einfachem Verschieben und Blättern können vergrößert, verkleinert und Funktions- oder Strukturknoten auf- und zugeklappt werden. Zur schnelleren Simulation ist es möglich Breakpoints zu setzen und die Anzeigegeschwindigkeit anzupassen. Außerdem kann die Ausführung Schritt für Schritt durchgespielt, unterbrochen oder zurückgesetzt werden.

Simulation mit Kommandozeile

Im Batch-Modus läuft das kompilierte Mitrion-C-Programm ohne Unterbrechung durch und gibt Ausgaben entsprechend im Quellcode angegebener Beobachtungspunkte (engl. *Watchpoints*) aus. Zusätzlich können auch Speicherzugriffe ausgegeben werden.

Im Gegensatz zum grafischen Debug-Modus ist die Simulation im Batch-Modus der konkreten Implementierung des MVP auf dem FPGA sehr nahe, auch wenn keine Hardware-Beschränkungen betrachtet werden. Damit ermöglicht diese Art der Simulation eine Abschätzung der Performance des Mitrion-C-Programmes auf dem FPGA. Eine entsprechende Konsolenausgabe am Ende des Simulationsdurchlaufs stellt die Anzahl der 100MHz Taktzyklen dar, welche benötigt werden um das Mitrion-C-Programm auf dem FPGA auszuführen.

```
ROOT
|  l: 50.0 ch: 50 rch: 50  BW limited. body ch: 50 rch: 50
   |--1
   | iterations: 5
   |   Env/main/foreach<5>
   |  l: 50.0 ch: 50 rch: 50  BW limited. body ch: 10 rch: 10
      |--1
      | iterations: 10
      |   Env/main/foreach<5>/for<10>
      |  l: 30.0 ch: 10 rch: 10  Latency limited dataloop. body latency: 3.0
```

Auflistung 4.2: Mitrion Bandbreitengraph

Zusätzlich gibt der Simulator einen Bandbreitengraph aus (vgl. Auflistung 4.2 und Quellcode 4.1). Dieser Graph verdeutlicht die Schleifenstruktur eines Mitrion-C-Programmes nach der Optimierung und zeigt die geschätzten Latenzen und Durchsätze der Schleifen an. Die Latenz einer Schleifenstruktur wird als Anzahl von Taktzyklen, bis die Schleife alle Ergebnisse produziert hat, angezeigt. Die Leerlaufzeit

(engl. *choke*) und damit die Anzahl der Taktzyklen, bis eine Schleife erneut ausgeführt werden kann, wird ebenso angezeigt. Dazu wird noch die durch andere Mitrion-C-Programmteile bedingte Leerlaufzeit (engl. *relaxed choke*) dargestellt. Damit ist eine Schleifenstruktur entweder durch deren Bandbreite oder Latenz begrenzt. Für eine bandbreitenbegrenzte Schleife werden *choke* und *relaxed choke* angezeigt. Für eine durch ihre Latenz beschränkte Schleife wird die zugehörige Schleifenlatenz angezeigt.

Da die Länge von Streams dynamisch ist, sind die Angaben bei Schleifen über Streams nicht deterministisch und können demnach nicht zur Performance-Abschätzung solcher Schleifen verwendet werden.

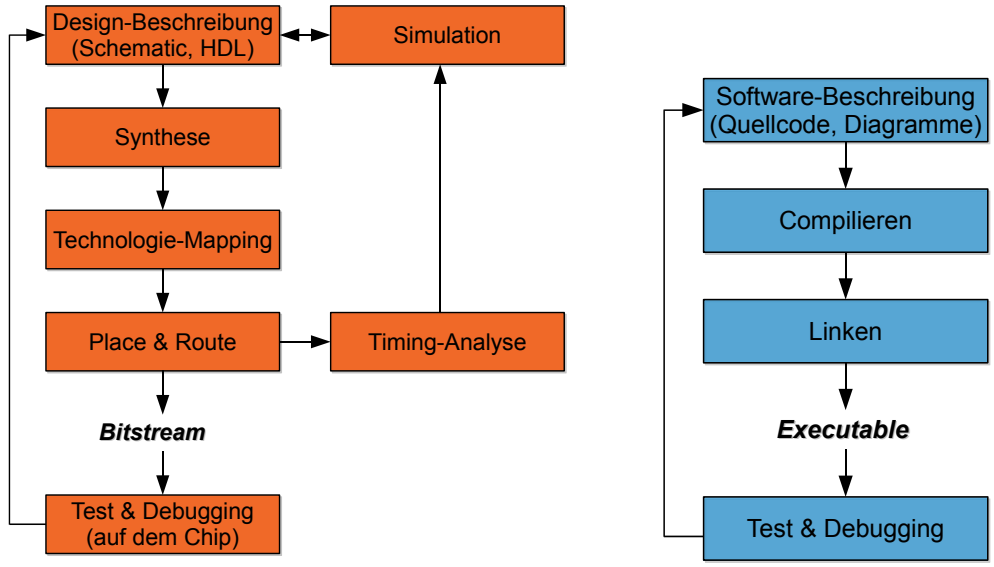
Simulation Server Mode

Im Server-Modus kann der Mitrion Simulator dazu genutzt werden, einen auf dem FPGA implementierten MVP zu emulieren. Aus einem auf dem Host ausführbaren ANSI-C- oder Fortran-Programm heraus wird der Simulations-Server über die Mithal API aufgerufen. Die Funktionsaufrufe sind dieselben, wie bei der Ausführung des MVP auf dem FPGA. Es müssen lediglich zwei Funktionsaufrufe angepasst werden, weil sich die Semantik einiger Argumente ändert. Wird der Server-Modus verwendet, müssen der Funktion *mitrion_fpga_allocate()* der Port und der Hostname des Simulations-Servers übergeben werden. Anstelle der Bitstream-Bezeichnung muss der Funktion *mitrion_processor_create()* im Server-Modus der Dateinamen des Mitrion-C-Quellcodes übergeben werden.

4.2 Hardware-Entwicklungsablauf

Die typische Hardware-Entwicklung beginnt mit einem Hardware-Entwurf. Dieser wird anschließend in eine Hardwarebeschreibungssprache (HDL) übertragen, simuliert und mittels automatisierter Vorgänge auf die Zieltechnologie umgesetzt. In dieser Arbeit ist das Ergebnis der Hardware-Entwicklung ein Bitstream, welcher die notwendigen Informationen enthält um den FPGA entsprechend der vorgenommenen Spezifikation zu programmieren. Auf die einzelnen Stufen der Hardware-Entwicklung wird im Verlauf dieses Kapitels noch genauer eingegangen (siehe Abschnitt 4.2.1).

Zwischen Hardware- und Softwareprogrammierung gibt es einige grundlegende Unterschiede. Ein Hardwaredesign ist die physische, ein Softwaredesign die logische Implementierung einer Funktionalität. Aus technischer Sicht bezeichnet Software alle nichtphysischen Funktionsbestandteile eines softwaregesteuerten Gerätes. Hardware gibt den physischen Rahmen vor, innerhalb dessen Grenzen eine Software funktioniert. Bei der Hardwareprogrammierung implementiert man für ein mikroelektronisches System einen bestimmten Umfang an Befehlen, welche durch digitale Schaltungen realisiert werden. Eine Abfolge von Befehlen, die sich in einem Speicher befinden, können als Software bezeichnet werden. In Abbildung 4.4



(a) Hardware-Entwicklungsablauf (b) Software-Entwicklungsablauf

Abbildung 4.4: Vergleich von Hardware- und Softwareentwicklung

sind Hardware- und Software-Entwicklung im Überblick dargestellt, wobei das Ergebnis ein Bitstream, zur Programmierung des FPGA, bzw. eine ausführbare Datei ist.

Die klassische Vorgehensweise bei der Softwareentwicklung beginnt mit der Beschreibung einer Funktionalittät oder einem Algorithmus in einer höheren Programmiersprache, wie C, C++, Fortran, usw. Nachdem der Compiler den Quellcode in Maschinensprache übersetzt hat, fügt der Linker einzelne Module zu einem ausführbaren Programm zusammen. Die korrekte Funktionsweise wird mit Hilfe von Testfällen überprüft. Bei einer falschen Ausgabe, wird der Fehler im Quellcode gesucht und beseitigt. Danach wird erneut übersetzt, gelinkt und getestet, solange bis alle Testbeispiele fehlerfrei durchlaufen.

Bei der Entwicklung von Hardware kann das zugehörige Design vor der eigentlichen Implementierung simuliert und damit auf Fehler untersucht werden. Während die Simulation bei der Hardware-Entwicklung zwingend erforderlich ist, ist sie bei der Software-Entwicklung überflüssig. Ein Grund dafür ist die Zeit, die für eine Übersetzung des Designs benötigt wird. Die Übersetzungszeiten für Software-Entwürfe sind in der Regel kurz. Die automatisierte Implementierung eines Hardware-Entwurfs benötigt dagegen wesentlich mehr Zeit. Für größere Designs sind Übersetzungszeiten von mehr als einem Tag realistisch.

Der Mitrion-Enwicklungszylkus (siehe Abbildung 4.1) ordnet sich zwischen Hardware- und Software-Entwicklung ein. Wegen der auch für Mitrion-C-Programme notwendigen, zeitintensiven Hardware-Synthese wird Simulation verwendet, um die Funktionalität der Beschreibung zu überprüfen. Allerdings findet diese in Mitrion-C mit typischen Mitteln der Softwareentwicklung statt (Watch-, Break-Points).

Der herkömmliche Ansatz zur Beschreibung von Hardware und damit auch FPGAs sind Hardwarebeschreibungssprachen (HDLs). Die am meisten genutzten HDLs sind Verilog und VHDL. In dieser Arbeit wurden alle Fallbeispiele (siehe Kapitel 5) in VHDL geschrieben, weshalb im Folgenden ausschließlich auf diese HDL eingegangen wird. Grundlagenwissen zum Entwurf mit VHDL wird in [Pre05] und [Mä08] vermittelt, worauf auch die folgenden beiden Abschnitte basieren.

Einer der wesentlich Gründe VHDL anstelle von Verilog zu verwenden, ist die größere Anzahl von Konstrukten zur High-Level-Modellierung (vgl. [Smi96]), wie z.B.:

- Anweisungen zur Erzeugung eigener abstrakter Datentypen,
- Bibliotheken mit Paketen zur Wiederverwendung,
- Konfigurationsanweisungen für die Struktur des Entwurfs,
- Generate-Anweisung um wiederkehrende Strukturen zu erzeugen,
- Anweisung zur Erstellung generischer Modelle mit individuellen Eigenschaften (z.B. Bitweiten).

Alle aufgezählten Konstrukte können für ein synthetisierbares Design verwendet werden. Verilog bietet außer der Parametrisierung und entsprechender Parameter-Überladung von Modellen keine zu VHDL vergleichbaren High-Level-Konstrukte.

4.2.1 Hardware-Entwurf mit VHDL

Die Sprache VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language) dient der Beschreibung und Simulation von Hardware und ist in Europa die am weitesten verbreitete HDL. Das Entwurfsziel kann programmierbare Logik, wie Complex Programmable Logik Devices (CPLDs) oder FPGAs oder festverdrahtete Logik, wie z.B. ASICs sein. Ursprünglich entwickelt um Testumgebungen für die Simulation integrierter Schaltungen zu entwerfen, deckt der Sprachumfang von VHDL heute alle während des Entwurfsvorgangs anfallenden Beschreibungen der Schaltung ab.

VHDL erlaubt die Beschreibung von der Systemebene bis hin zur Gatterebene. Digitale Schaltungen bestehen aus Kombinatorik und Speicherelementen. Um beides für reale Hardware nachzubilden ist die grundlegende Denkweise sehr verschieden zu üblichen Programmiersprachen, wie z.B. C oder Java. VHDL ist eine datengetriebene Sprache, in der nicht wie bei anderen prozeduralen Programmiersprachen die Reihenfolge der Befehlsausführung im Mittelpunkt steht.

Komplexe Designs werden durch strukturellen Entwurf hierarchisch aufgebaut. Dies geschieht bei VHDL über Komponenten, welche Teil einer übergeordneten Struktureinheit sind. Die Komplexität einzelner Einheiten (engl. entities) kann von einfachen Gattern bis hin zu vollständigen Prozessorkernen reichen.

Sie können Verhalten-, Datenfluss- und Strukturbeschreibungen beinhalten, wobei in VHDL Schnittstelle und Implementierung strikt voneinander getrennt werden.

Eine VHDL-Implementierung besteht im wesentlichen aus einer Menge von Prozessen, die gleichzeitig abgearbeitet werden und nebenläufigen Anweisungen, welche kombinatorische Logik beschreiben. Prozesse bilden komplexe nebenläufige Anweisungen, wobei die Reihenfolge der Anweisungen innerhalb eines Prozesses von Bedeutung ist. Es wird zwischen kombinatorischen und synchron getakteten Prozessen unterschieden. Kombinatorische Prozesse beschreiben das Verhalten von Schaltnetzen und haben eine Sensitivitätsliste, welche die Eingangssignale des Netzes beinhaltet. Ändert sich eines dieser Signale, wird der Prozess angestoßen und es können sich Signale ändern, welche wiederum weitere Prozesse anstoßen. Synchron getaktete Prozesse haben in der Sensitivitätsliste das Taktsignal (und das Reset-Signal, falls ein asynchrones Reset benötigt wird). Sie beschreiben Zustandsübergänge von Registern und legen damit die Werte von Signalen fest, die zur Taktflanke übernommen werden sollen.

VHDL ist eine typenorientierte Programmiersprache. Zudem werden grundsätzlich drei Klassen von Informationsträgern unterschieden: Konstanten, Signale und Variablen. Einer Konstanten kann üblicherweise nur einmal ein Wert zugewiesen werden. Dieser ist dann in der gesamten Einheit bekannt und unveränderlich. Signale dienen der Informationsübermittlung zwischen einzelnen nebenläufigen Funktionsblöcken und verknüpfen damit Komponenten und Prozesse untereinander. Ihr Zustand ändert sich im Vergleich zu den Variablen immer erst am Ende eines sequentiellen Prozesses. Variablen können ausschließlich innerhalb von sequentiellen Anweisungsfolgen, wie Prozessen und Prozeduren, verwendet werden. Wie bei typischen Programmiersprachen wirken Variablenzuweisungen ohne Verzögerung.

Mit den sog. *Generics* wird ein generischer Hardwareentwurf ermöglicht. Als Paramter einer Schnittstelle lassen sich damit Entwürfe schnell und einfach anpassen. Das *Generate*-Statement erlaubt die Beschreibung sich wiederholender oder bedingter Strukturelemente. In Mitrion-C lässt sich eine Parametrierung des Entwurfs über *define*-Anweisungen lösen.

VHDL benutzt ein Bibliothekskonzept um den Zugriff auf gemeinsame Datenbestände zu ermöglichen, Entwürfe wiederzuverwenden und herstellerspezifische Bibliotheken einzubinden. Pakete sind Bestandteile von Bibliotheken und können lokale Typen und Konstanten, Komponentendeklarationen, Funktionen und Prozeduren bündeln. Sie werden wie Einheiten in Schnittstelle und Implementierung untergliedert. Ein mächtiges Hilfsmittel für Verhaltensmodelle sind Funktionen und Prozeduren, welche jedoch nur in beschränktem Umfang in der Synthese unterstützt werden.

Eine Möglichkeit den Hardware-Entwurf großer System (z.B. SoCs) zu vereinfachen und zu beschleunigen sind IP-Cores (**I**ntellectual **P**roperty Cores). In der Halbleiterindustrie werden damit fertige Entwürfe bezeichnet, welche in anderen Entwürfen erneut verwendet werden können (Design Reuse). Kürzere Ent-

wicklungszeiten führen meist auch zu geringeren Kosten. Es wird zudem zwischen Soft-IP-Cores und Hard-IP-Cores unterschieden. Hard-IP-Cores sind als fertige Schaltung herstellerseitig unveränderbar in den Chip des FPGAs integriert (siehe z.B. Abschnitt 2.4). Soft-IP-Cores hingegen liegen typischerweise als generische Netzlisten oder aber HDL-Quellcode vor. Diese Darstellung als Netzliste schützt den Anbieter vor Reverse Engineering, verhindert allerdings auch die Portierung auf beliebige Zielplattformen. Daher bieten die FPGA-Hersteller in ihren Entwicklungsumgebungen IP-Core-Generatoren zur Erzeugung von Netzlisten für verschiedene FPGA-Modelle an. Zudem gibt es Unternehmen, die sich darauf spezialisiert haben, Teile oder komplette integrierte Schaltkreise zu entwerfen und Lizenzen dieser Designs zu verkaufen. Typische Beispiele für Soft-IP-Cores sind Mikrokontroller wie PicoBlaze. Synthetisierbare IP-Cores, die im HDL-Quellcode vorliegen, ermöglichen dem Benutzer Anpassungen auf der funktionalen Ebene vorzunehmen und können sowohl für FPGAs als auch ASICs benutzt werden. Beide Formen von Soft-IP-Cores werden im frei programmierbaren Bereich eines FPGAs implementiert. Ein IP-Core im Schaltkreisentwurf ist vergleichbar mit einer Bibliothek in der Hochsprachenprogrammierung.

Es gibt zwei typische in VHDL verwendete Programmierstile. Häufig wird der Datenflussstil verwendet (auch in den Fallbeispielen dieser Diplomarbeit). Hierbei werden pro Komponentenarchitektur eine Vielzahl parallel ablaufender (meist kleiner) synchron getakteter Prozesse und nebenläufiger Anweisungen beschrieben. Mit den Prozessen werden einzelne, meist elementare Hardwarestrukturelemente direkt in VHDL abgebildet. Signale dienen sowohl zur Zustandsspeicherung innerhalb der einzelnen Prozesse, als auch zur Informationsübermittlung zwischen Prozessen und Komponenten. Die sog. 2-Prozessmethode findet vor allem im synchronen FPGA-Design Verwendung. Die Komponentenarchitektur besteht dann nur noch aus zwei Prozessen: Ein meist sehr komplexer rein kombinatorischer Prozess, welcher den kompletten Algorithmus und das Verfahren beinhaltet und ein recht einfacher getakteter Prozess, welcher alle Register (Zustandsspeicher) beinhaltet.

Darüber hinaus gibt es sprachliche Erweiterungen in Form von VHDL-AMS, mit welchen auch analoge Systeme beschrieben werden können. Diese soll den Rahmen elektrischer Simulation verlassen und auch mechanische Elemente, Sensoren und Aktoren modellieren, um zu einer möglichst vollständigen Systemsimulation zu gelangen.

Mit VHDL ist es auch möglich, nicht synthesefähigen, funktionalen Code zu schreiben. Einige Anweisungen sind nur für die Simulation verwendbar und können nicht in Hardware umgesetzt werden. Nicht synthesefähiger Code dient damit der Schaltungssimulation und wird in sog. Testbenches beschrieben, um die Umgebung der zu synthetisierenden Hardware zu simulieren. Damit kann dann das Verhalten von z.B. Schnittstellenprotokollen getestet werden.

4.2.2 Simulation

Die Simulation ist eine Vorgehensweise zur Analyse von Systemen, die für eine theoretische oder formelmäßige Behandlung zu kompliziert sind. Dies ist überwiegend bei dynamischem Systemverhalten der Fall. Weitere Gründe für den Einsatz einer Simulation können z.B. sein, wenn die Untersuchung am realen System entschieden aufwändiger wäre oder das reale System (noch) nicht existiert. Jede Form der Simulation hat natürlich auch Grenzen, wie z.B. die Begrenztheit der Mittel (Rechenkapazität, Zeit, etc.). Aufgrund dieser Einschränkungen muss ein Modell möglichst einfach sein, was bedeutet, dass eine grobe Vereinfachung der Realität vorgenommen wird. Dies beeinflusst wiederum die Genauigkeit der Simulationsergebnisse.

Diese anwendungsunspezifische Betrachtung kann direkt in die Hardware-Simulation übernommen werden. Auch wenn es sich um rekonfigurierbare Hardware wie FPGAs handelt, ist die Simulation bei der Hardware-Entwicklung ein nicht zu unterschätzender Aufwand. Sie wird hauptsächlich deswegen verwendet, weil die Fehlersuche auf dem FPGA kompliziert ist und eine Hardware-Synthese, je nach Größe der Schaltung, mehrere Tage dauern kann. Ohne Simulation würde sich der Hardware-Entwicklungszyklus stark verlängern.

Für die Simulation paralleler Hardware ist es notwendig die Abläufe auf ein sequentielles Programm abzubilden. Dies wird mit sog. Zeitscheiben vorgenommen. Es werden alle nebenläufigen Anweisungen und Prozesse einzeln ausgeführt und jedem Signalwert ein Ereignis zugeordnet. Innerhalb einer Zeitscheibe hat jedes Signal einen festen Wert. Da es sich um eine ereignisgetriebene Simulation von Signalen handelt, brauchen für eine nachfolgende Zeitscheibe nur Prozesse betrachtet zu werden, in denen sich ein Signal aus ihrer Sensitivitätsliste oder einer wait-Anweisung geändert hat. Sowohl die Sensitivitätsliste, als auch wait-Anweisungen bewirken, dass Anweisungen innerhalb eines Prozesses sequenziell abgearbeitet und dadurch neue Ausgangswerte erzeugt werden.

Bei der Simulation von Hardware werden prinzipiell zwei Simulationsarten unterschieden. Die reine Verhaltenssimulation (engl. behavioral simulation) der beschriebenen Hardware, in der die funktionalen Zusammenhänge der Schaltung geprüft werden und die Simulation der fertig platzierten und verdrahteten Schaltung (engl. timing simulation). Bei letzterer wird die Schaltung zunächst synthetisiert, platziert und verdrahtet, bevor aus der fertigen Schaltungsanordnung (Netzliste mit Informationen über die Verzögerungen) die zugehörigen Laufzeitinformationen entnommen werden. Der Vorteil besteht hier in dem genaueren Modell, um beispielsweise Zeitablaufprobleme in der Zielhardware bereits während der Simulation erkennen zu können. Der wesentliche Nachteil besteht in dem damit verbundenen hohem Rechenaufwand und die im Vergleich zur Verhaltenssimulation lange Simulationszeit.

Weiter kann zwischen Testbenches (Simulationsbeschreibung) mit integrierter Fehlererkennung und Testbenches, welche nur den reinen Zeitverlauf (sog. *Waveforms*) darstellen, unterschieden werden. Der Nachteil letzterer ist, dass dabei Fehler in der Implementierung durch die manuelle Prüfung leicht übersehen werden können, vorallem bei komplexen Schaltungen.

4.2.3 Hardware-Synthese

Als Synthese (aus dem Griechischen sýnthesis - die Zusammensetzung, Zusammenfassung, Verknüpfung) bezeichnet man den Umsatz (die Vereinigung) von zwei oder mehr Elementen (Bestandteilen) zu einer neuen Einheit. Anders formuliert versucht die Synthese aus mehreren Komponenten und/oder Anweisungen ein System zusammenzusetzen. Im digitalen Schaltkreisentwurf wird der Begriff der Synthese verwendet, um die Umwandlung einer funktionalen Beschreibung einer Schaltung in eine Gatter-Netzliste zu beschreiben.

Ein Hardware-Synthese-Schritt kann als eine automatisierte Implementierung einer formalen Spezifikation auf einer niedrigeren Abstraktionsebene betrachtet werden (vgl. [Hoc08]). Mit niedriger werdender Abstraktionsebene kommen Informationen hinzu, wobei ab der Register-Transfer-Ebene technologiespezifische Daten mit einfließen. Abbildung 4.5 zeigt die verschiedenen Abstraktionsebenen (von oben nach unten fallender Abstraktionsgrad) und die Hardware-Synthese-Schritte um eine niedrigere Abstraktionsebene zu erreichen.

Sowohl für die aus einem Mitrion-C-Programm und der MVP-Architektur synthetisierte VHDL-Beschreibung, als auch für handgeschriebenen VHDL-Code muss eine Hardware-Synthese durchgeführt werden, um eine der Beschreibung entsprechenden Implementierung zu erhalten.

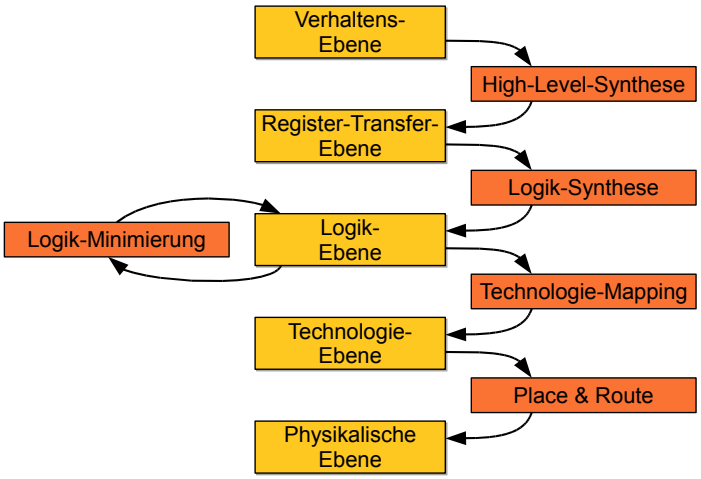


Abbildung 4.5: Hardware-Synthese im Überblick ([Hoc08], Kapitel 1)

High-Level-Synthese

Dieser erste Synthese-Abschnitt zerlegt die Verhaltensbeschreibung in Daten- und Kontrollpfad. Es werden im Wesentlichen drei Vorgänge durchgeführt:

1. Allokation - Ermittlung der zu verwendenden Ressourcen
2. Binding - Zuordnung von Operationen und Variablen zu Ressourcen
3. Ablaufplanung (engl. Scheduling) - Planung des zeitlichen Ablaufs aller Operationen

Logik-Synthese und Logik-Minimierung

Ein weiterer Synthese-Schritt ist die Umwandlung der Komponenten auf Register-Transfer-Ebene in Schaltfunktionen (boolesche Funktionen). Wesentliche Aufgaben sind hier:

- Zustandskodierung von endlichen Automaten (FSM) (bei FPGAs meist One-Hot-Kodierung)
- Dekomposition von FSMs (Umwandlung komplexer FSMs in eine Komposition einfacher FSMs, welche einfacher zu implementieren sind)
- Darstellung von FSMs durch Übergangslogik und Zustandsregister
- Abbildung der Datenpfad-Komponenten auf Schaltfunktionen

Während der Logik-Minimierung wird schließlich versucht eine minimale Implementierung gegebener Schaltfunktionen zu erreichen und damit eine kostengünstige Realisierung einer Menge von booleschen Funktionen zu finden. Da exakte Verfahren, wie z.B. der Quine-McCluskey-Algorithmus NP-vollständig sind, werden an dessen Stelle Heuristiken verwendet. Andernfalls wären große Schaltungen nicht in annehmbarer Zeit synthetisierbar. Die graphische Methode zur Logik-Minimierung mittels eines Karnaugh-Veitch-Diagramms ist für die Berechnung auf einem Computer ungünstig und wird demnach nicht für den automatisierten Entwurf verwendet. Für FPGAs ist die Aufgabenstellung noch komplexer, da eine Funktion mit verschiedenen Grundenelementen des FPGAs realisiert werden kann und damit die Anzahl der möglichen Realisierungen steigt.

Technologie-Mapping

An dieser Stelle werden die bereits minimierten Schaltfunktionen, Register und andere Makro-Module auf die Zieltechnologie abgebildet. Dabei werden die Schaltfunktionen in Elemente der Zieltechnologie zerlegt. Für ein „Full-Custom-Design“ wird üblicherweise „Library-Binding“ (Auswahl von Elementen aus einer Bibliothek, welche die gewünschte Funktion haben) verwendet, was bei FPGAs allerdings

nicht anwendbar ist. Anstelle dessen werden algebraische Ansätze und die funktionale Dekomposition genutzt.

Plazierung und Verdrahtung

Während der Plazierung wird die Anordnung der Technologieelemente auf dem Chip vorgenommen, d.h. für FPGAs werden den physisch vorhandenen CLBs die zu realisierenden Funktionen zugeordnet. Als Eingabe wird die Netzliste der Schaltung mit den Informationen aus dem Technologie-Mapping verwendet. Randbedingungen, wie z.B. Verdrahtung und Ein- und Ausgänge der Schaltung müssen hier mit betrachtet werden. Auch dieser Vorgang ist ein NP-vollständiges Problem und kann nur bei gleichzeitiger Verdrahtung exakt gelöst werden. Die Verdrahtung selber ist stark abhängig von der Zieltechnologie und der Plazierung. Bei FPGAs wird diese über verschiedene Verbindungskanäle vollzogen, während bei ASICs Metalllagen genutzt werden. Die Verdrahtung hat zudem maßgeblichen Einfluss auf die Performance einer Schaltung.

Das sog. *Place & Route* ist sehr rechenintensiv und kann bei großen Designs mit harten zeitlichen Bedingungen viele Stunden bis Tage in Anspruch nehmen. Der Zeitaufwand hängt dabei nicht nur von der Komplexität des FPGA-Designs ab, sondern auch vom Grad der gewählten Optimierung. Verbindungswege müssen möglichst kurz sein, um die Signallaufzeiten und den Flächenbedarf für benutzte Verdrahtung gering zu halten. Mitunter werden auch CLBs verwendet, um eine Verdrahtung unter entsprechenden zeitlichen Bedingungen zu ermöglichen.

Nach der Verdrahtung sind die Signallaufzeiten fest und können für eine Timing-Simulation verwendet werden. Während der Timing-Analyse werden zuvor noch der längste Pfad und die damit höchste zu erreichende Taktfrequenz ermittelt.

5 Fallbeispiele

Nachdem die Hardware und ihre Kenndaten bereits in den vorangegangenen Kapiteln vorgestellt wurden, soll nun die Leistungsfähigkeit der SGI RASC Programmierplattform anhand von Benchmarks und Beispielanwendungen bewertet werden. Hierfür wurden die in den Abschnitten 3.3 vorgestellten Kommunikationsvarianten implementiert und anschließend Bandbreiten- und Latenzmessungen vorgenommen. Mit den Implementierungen des MD5-Algorithmus und des Damenproblems sollen neben dem Leistungsvermögen FPGA-basierter Beschleuniger auch die Performance- und Implementierungsunterschiede zwischen Mitrion-C und VHDL verdeutlicht werden. Da Mitrion nur die Ansteuerung eines Koprozessors unterstützt, wird aus Gründen der Vergleichbarkeit auch nur ein RC100-FPGA betrachtet. Dementsprechend verdoppelt sich der gemessene Geschwindigkeitsgewinn für ein RC100-Blade, wenn zwei Prozesse gestartet und damit die FPGAs unabhängig voneinander angesteuert werden.

In dieser Arbeit wurden die *Mitrion SDK XL v1.5.3 build 535* und *Xilinx ISE 9.2.04i* verwendet. Bei der Betrachtung von Mitrion-C-Quellcode ist zu beachten, dass der Mitrion-Compiler Optimierungen vornimmt und damit eine sichere Bestimmung der benötigten FPGA-Takte pro Zuweisung nicht möglich ist. Anstelle dessen können die Ausgaben des Kommandozeilen-Simulators herangezogen werden.

Bei der Programmierung von RASC unter Verwendung einer Hardwarebeschreibungssprache müssen im Gegensatz zu Mitrion-C-Implementierungen zusätzlich die Verilog-Wrapper-Module und die Konfigurationsdateien erzeugt werden (z.B. mit dem Algorithmus-Konfigurations-Tool, siehe Abschnitt 3.4.4). Wird wie in dieser Arbeit VHDL verwendet, muss in der Schnittstelle zum Algorithmus (Verilog-Modul) anschließend noch die Verdrahtung zu den VHDL-Komponenten vorgenommen werden. Um zu spezifizieren, welche Kommunikationsressourcen genutzt werden, müssen zusätzlich zu den Konfigurationsdateien die Extractor-Anweisungen als Kommentare im Quellcode erstellt werden. Bei Mitrion-C besteht die Schnittstelle zu den Core Services aus Über- und Rückgabeparametern der main-Funktion. Die Wrapper-Module und Extractor-Anweisungen werden daraus automatisch erzeugt.

Nach der Beschreibung des Algorithmus folgt die Hardware-Synthese, welche je nach betriebenem Aufwand zu unterschiedlichen Ergebnissen führt. Komplexere Optimierungsvorgänge können mit einer größeren Wahrscheinlichkeit die Entwurfsanforderungen erfüllen, beanspruchen jedoch mehr Zeit. Mitrion erlaubt die Einstellung des Hardware-Synthese-Aufwandes in drei Stufen. Vor der Generierung des

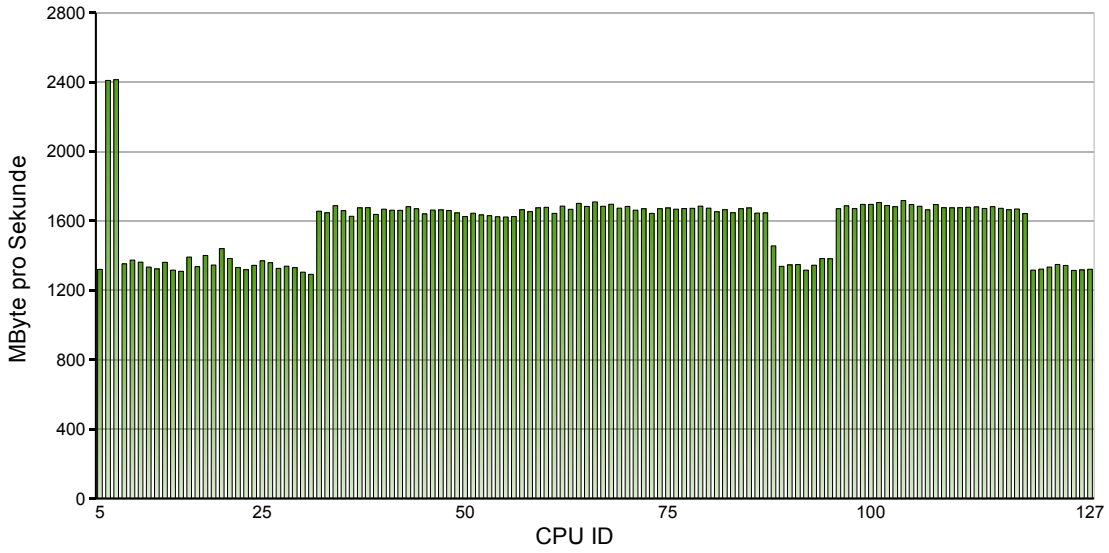


Abbildung 5.1: Direct IO Streaming Durchsatz aller Knoten (256 MiByte)

VHDL-Designs werden allerdings noch die voraussichtlich benötigten FPGA-Ressourcen abgeschätzt und ggf. die Hardware-Umsetzung abgebrochen. Während der Tests fiel auf, dass Mitrion 200448 anstelle von 178176 auf dem Virtex-4 LX200 verfügbaren Flipflops für die Abschätzung verwendet (aus Mitrion Debug-Ausgaben), die prozentualen Angaben der realen Umsetzung jedoch recht nahe kamen. Bei der Verwendung einer HDL kann die Hardware-Synthese, über die vom Algorithmus-Konfigurations-Tool erzeugten und voreingestellten Konfigurationsdateien, angepasst werden. Damit können für die Synthese, das Technologie-Mapping und die Plazierung und Verdrahtung unabhängig voneinander Einstellungen vorgenommen werden, welche mitunter starken Einfluss auf die Umsetzung in die entsprechende Zieltechnologie haben können. Dieses Optimierungspotenzial kann auch mit Mitrion genutzt werden, indem die generierten Dateien angepasst und anschließend mit der RASC-Tool-Chain verarbeitet werden.

Um die Datendurchsatzmessungen der Mitrion- und VHDL-Implementierungen gut vergleichen zu können, wurde zu jeder Kommunikationsvariante ein Host-Programm mit dem RASC-API (siehe 3.4.1) geschrieben, welchem über Parameter die zu verwendende Implementierung und die Größe der zu sendenden Daten mitgeteilt werden. Große Übertragungsraten und geringe Latenzzeiten werden genau dann erreicht, wenn das Host-Programm auf einem zum RC100-Blade topologisch gesehen nahen CPU ausgeführt wird. Mit *dplace*[1] wurde das Host-Programm mit einer festen Datenmenge auf allen frei verfügbaren CPUs (124) ausgeführt und die schnellste Verbindung ermittelt (vgl. Abbildung 5.1). Die Messungen der Transferraten für verschiedene Datenmengen wurden dann bei geringer Auslastung des Systems in einem Durchlauf immer auf der gleichen CPU vorgenommen.

Bei den Messungen wurden maximal 256 MByte (entspricht der Größe einer Hugepage) allokiert und

[1] Tool zum Binden von Prozessen an CPUs

diese mehrfach für die Übertragung größerer Datenmengen an einen FPGA des RC100-Blades geschickt. Im RASC-Benutzerhandbuch ([SGI08]) wird empfohlen, Daten mit der Größe einer Hugepage zu schicken, um maximale Datenraten zu erreichen.

5.1 SRAM-Schnittstelle

Die Kenndaten und Konfigurationsmöglichkeiten des SRAM-Interfaces wurden bereits in Abschnitt 3.3.1 vorgestellt. Mitrion nutzt als minimale Implementierung der Kommunikation zwei 16 MByte Speicherbänke mit 128-Bit-Wörtern (Konfiguration (1)). Optional kann noch die verbliebene 8 Mbyte SRAM-Bank hinzugenommen werden. Andere Konfigurationen, wie fünf unabhängige SRAM-Bänke oder kein SRAM-Speicherkontroller, werden nicht unterstützt.

Um den maximalen Durchsatz für die Kommunikation zwischen Host und Algorithmus über die SRAM-Schnittstelle zu ermitteln, bietet es sich an die Konfiguration (1) zu implementieren, was sowohl in VHDL als auch Mitrion-C vorgenommen wurde. Durch das Zusammenfassen von zwei SRAM-Bänken (je 8 MByte und 3,2 GByte/s Bandbreite) verdoppelt sich die nutzbare Bandbreite an dieser Schnittstelle auf 6,4 GByte/s und ist theoretisch kein Flaschenhals mehr, was in Messungen auch bestätigt werden konnte. Zudem wird bei der VHDL-Implementierung die Latenz von indizierten SRAM-Speicherzugriffen aus dem Algorithmus-Block gemessen. Weiterhin wird zwischen *Buffered I/O* und *Direct I/O* unterschieden (vgl. Abschnitt 3.4.1 - Absätze drei und vier).

5.1.1 Implementierung

Das einfache Mitrion-C-Programm aus 5.1 verwendet den RC100-SRAM zum Datentransfer zwischen Host und Algorithmus. Um auf Speicher-Blöcke oder -Bänke zugreifen zu können werden bei Mitrion-C an diese gebundene Instance-Tokens verwendet. Die SRAM-Bänke werden in Form solcher Token der main-Funktion übergeben und mit dem Schlüsselwort *mem* versehen. Eine Größenangabe zu diesen legt fest, wieviele Werte vom Algorithmus gelesen bzw. geschrieben werden dürfen. Für eine 16 MByte SRAM-Bank können minimal 2048 und maximal 1048576 128-Bit-Werte übertragen werden. Die untere Grenze ist von Mitrion festgelegt, während sich die obere Grenze aus der Größe der Speicherbank ergibt. Die Daten werden auch übertragen, wenn der Algorithmus diese nicht verarbeitet. Um Multibuffering (siehe 3.3.1 - letzter Absatz) zu ermöglichen, wird die SRAM-Bank in Segmente unterteilt. Die Beispiel-Implementierung verwendet zwei Segmente pro 16 MByte SRAM-Bank, wodurch immer ein Vielfaches von 524288 128-Bit-Werte pro Richtung zwischen Host und FPGA übertragen werden muss. Über eine foreach-Schleife werden die Werte verarbeitet, wobei die Funktionen *memread()* und *memwrite()* den

```
0  Mitrion-C 1.5;
   // Options: -cpp -alg-name testsramitc -alg-id 4242 -alg-ver 2

   #define SIZE 524288
   #define SRAM mem bits:128[SIZE]
5
   (SRAM, SRAM) main(SRAM sram0, SRAM sram1, uint:64 inc_val){
     (sram0_done, sram1_done) = foreach(i in <0 .. SIZE-1>){
       (in128, sram0_rd) = memread(sram0, i);
         uint:16[8] in16 = in128;
10       uint:16 in16_0 = in16[0] + inc_val;
         uint:16 in16_4 = in16[4] + inc_val;
         bits:128 result = [in16_0, in16[1], in16[2], in16[3],
                            in16_4, in16[5], in16[6], in16[7]];
         sram1_wr = memwrite(sram1, i, result);
15     } (sram0_rd, sram1_wr);
   } (sram0_done, sram1_done);
```

Auflistung 5.1: Mitrion-C-Programm zum Test der SRAM-Anbindung

indizierten Zugriff auf den SRAM ermöglichen. Zwei 16-Bit-Additionen auf den oberen und unteren 64 Bit der übergebenen Werte werden als einzige Datenmanipulation durchgeführt und dienen später der Überprüfung, ob die Daten durch den FPGA korrekt verarbeitet wurden. Hierzu wird der aus dem SRAM gelesene 128-Bit-Wert in einen Vektor aus acht 16-Bit-Werten unterteilt und die unteren 16 Bit eines aus dem Host-Programm übergebenen Parameters (Nutzung eines *Algorithm Defined Register*) auf das erste und fünfte Vektorelement addiert. Die Reduzierung der Addition auf 16 Bit vereinfacht die Verdrahtung des Algorithmuskerns. Schließlich werden noch die verarbeiteten Instance-Tokens zurückgegeben. Das Mitrion-C-Programm besteht damit aus einer durch ihre Bandbreite begrenzten Schleife (Mitrion Bandbreitengraph), wobei jedoch die Speicherzugriffe zu einer sequentiellen Verarbeitung der Daten führen.

Die VHDL-Umsetzung des Datentransfers über den RC100-SRAM ist im Wesentlichen deswegen aufwendiger als in Mitrion-C, weil der Kontrollfluss zwischen Core Services und Algorithmus zusätzlich implementiert werden muss. Ein Ausschnitt des VHDL-Quellcodes ist im Anhang A.1 gelistet. Nicht enthalten sind die Schnittstelle und die Signaldefinitionen, sowie ein Prozess zur Messung der Latenz von indizierten Speicherzugriffen. Der erste Prozess sendet Leseanfragen (für die SRAM-Bank 0) an den Speicherkontroller, sobald dieser signalisiert, dass er bereit ist. Dabei werden gelesene Worte gezählt und beim Erreichen der vom Host-Programm übermittelbaren Grenze (*Algorithm Defined Register*) keine weiteren Anfragen mehr gesendet. Der zweite Prozess nimmt wie bei der Mitrion-C-Implementierung zwei 16-Bit-Additionen vor und schreibt die Werte in SRAM-Bank 1. Das Offset (Zeilen 16 und 42) für die Adressierung des Speichers wird durch die Core Services gesetzt und ermöglicht Multibuffering. Im Unterschied zur Mitrion-C-Implementierung muss die Anzahl der zu verarbeitenden Wörter vom Host-Programm angegeben werden. Dies erhöht die Flexibilität und erlaubt es Datenpakete kleiner

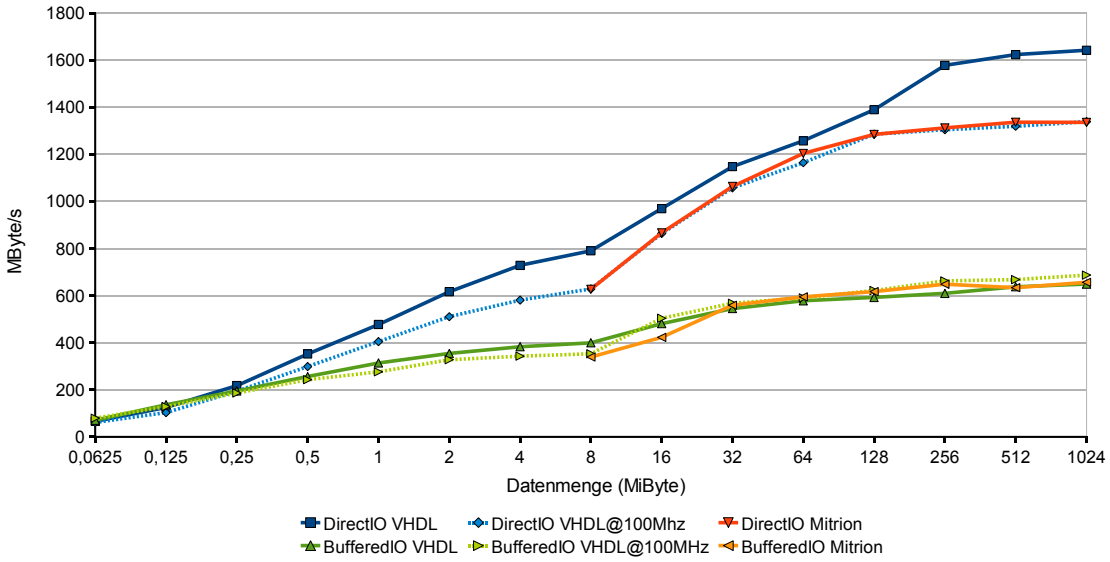


Abbildung 5.2: Durchsatz über den SRAM des RC100-Blades

als ein Multibuffering-SRAM-Segment zu senden. Dafür muss auch die Bitstream-Konfigurationsdatei angepasst werden, was aber ohne erneute Hardware-Synthese möglich ist.

5.1.2 Tests und Messungen

Die Hardware-Synthese konnte für 100 MHz Taktfrequenz des Algorithmus (VHDL und Mitrion) mit geringem Aufwand durchgeführt werden. Für 200 Mhz (nur VHDL) musste der Aufwand jedoch erhöht werden, um das Timing der Core Services zu ermöglichen. Das fertige Design benötigt bei gleichen Parametern der technologiespezifischen Synthese mit Mitrion 14% und mit der VHDL-Implementierung 12% der Slices des Virtex-4-LX200-FPGAs. Abbildung 5.2 zeigt eine Übersicht der Messungen zur SRAM-Kommunikation.

Die Datendurchsatzmessungen in [IJ08] wurden mit einer früheren Version von Mitrion vorgenommen und weisen bei 64 MiByte Paketgröße mit Direct I/O höhere und mit Buffered I/O niedrigere Datendurchsätze auf. Diese können jedoch je nach CPU-Wahl und Auslastung des Systems um einige hundert MByte/s abweichen. Damit erreichen die Messungen gerade die Hälfte des theoretische Maximums von 3,2 GByte/s. Auch wenn die Daten ausschließlich vom Host in den SRAM des RC100-Blades kopiert und anschließend wieder zurückgespielt werden, übersteigen die Datendurchsätze 1,65 GByte/s nicht. Dieser Test ist möglich, indem der VHDL-Implementierung vom Host-Programm als Parameter für die zu verarbeitenden Wörter null übergeben wird. Die Speicherbandbreite zwischen SRAM und Algorithmus und die Verzögerung des Algorithmus selber haben dann keinen Einfluss mehr auf den Datendurchsatz zwischen Host und FPGA. Dementsprechend ergaben Messungen für kleine Datenmengen eine Erhöhung des Durchsatzes, der jedoch mit zunehmenden Größe der zu übertragenden Daten vernachlässigbar wird.

Allgemein ist festzustellen, dass die Werte zwischen Messungen bei der Verwendung gleicher Parameter mitunter um bis zu 150 MByte/s schwanken.

Der Mitrion Virtual Processor wird mit 100 MHz getaktet, weshalb der maximale Datendurchsatz auf 1,6 GByte/s begrenzt ist. Die gemessenen Werte kommen damit durchaus in die Nähe des theoretischen Maximums. Die Ausführung mit dem VHDL-Design auf 100 MHz bestätigen die Datenübertragungsraten der Mitrion-Umsetzung. Zudem konnten mit dieser Implementierung Werte kleiner als acht MiByte (ein SRAM-Bank-Segment) aufgenommen werden.

Aus dem Diagramm kann die Größe des SRAM-Segments von acht MiByte abgelesen werden. Datenmengen über die Segment-Größe nutzen Multibuffering. Mit 256 MiByte (Größe einer Hugepage) ist die maximale Datenrate erreicht. Die Werte für größere Datenmengen wurden durch das erneute Senden des bereits allokierten Speichers ermittelt und bieten dementsprechend keine oder nur geringfügig höhere Datenraten. Die Halbierung der Segment-Größe einer SRAM-Bank führte zu einer leichten Verringerung des Datendurchsatzes.

Neben einer Latenz von 16 Taktzyklen bei 200 MHz Taktfrequenz (80 ns) wurde die Bandbreite mit 6,4 GByte/s (entspricht dem theoretischen Maximum) zwischen FPGA-Algorithmus und einer 16 MiByte SRAM-Bank gemessen. Für die Messung der Bandbreite wurden gleichzeitig 8 MiByte in den SRAM geschrieben und gelesen und die benötigten Takte gezählt. Die Latenz wurde für einen indizierten Speicherzugriff ebenso durch Zählen der Takte ermittelt.

5.2 Direct Streaming

Das direkte Streaming ist immer dann gut geeignet, wenn der Algorithmus keinen indizierten Zugriff auf einzelne Werte benötigt und Datenströme direkt verarbeiten kann oder die Block-RAM-Ressourcen des FPGA als Pufferung ausreichen. Zudem verwenden bei dieser Kommunikationsvariante die Core Services am wenigsten FPGA-Logik und es werden die höchsten Datendurchsätze erreicht.

5.2.1 Implementierung

Für Mitrion-C ist die Implementierung des direkten Streamings recht einfach. Es müssen neben den Speicherports ausschließlich der Übergabe- und Rückgabe-Stream in der Main-Funktion angegeben werden. Der Speicherkontroller der Core Services wird damit auch integriert, wenn der Algorithmus nicht auf den SRAM zugreift. Innherhalb des Mitrion-C-Programmes können unter Verwendung einer FOR- oder FOREACH-Schleife die Elemente des Streams verarbeitet werden. Während die RASC Core Services

```
Mitrion-C 1.5;
// Options: -cpp -alg-name teststrmitc -alg-id 4242 -alg-ver 2

#define SRAM mem bits:128[2048]

(SRAM, SRAM, bits:128(..))main(SRAM sram0, SRAM sram1,
                               bits:128(..) strm_in, uint:64 inc_val){
    strm_out = foreach(e in strm_in){
        uint:16[8] in16 = e;
        uint:16 in16_0 = in16[0] + inc_val;
        uint:16 in16_4 = in16[4] + inc_val;
        bits:128 result = [in16_0, in16[1], in16[2], in16[3],
                           in16_4, in16[5], in16[6], in16[7]];
    } result;
} (sram0, sram1, strm_out);
```

Auflistung 5.2: Mitrion-C-Programm zum Test des Direct Streaming

vier Streams je Richtung (Eingang, Ausgang) unterstützen, kann mit Mitrion-C nur ein Eingangs- und ein Ausgangs-Stream verwendet werden. Der Quellcode in 5.2 zeigt ein einfaches Beispiel zur Nutzung des direkten Streamings. Dabei wird, wie bei dem Beispiel für die SRAM-Kommunikation, jeder 128-Bit-Wert in zwei 64-Bit-Werte aufgeteilt und zu diesen ein vom Host-Programm übergebener Wert addiert. Um Verdrahtungsprobleme oder zusätzliche Verzögerungen von vornherein zu vermeiden, wurden die Additionen wiederum auf 16 Bit beschränkt.

Auch in VHDL ist das Streaming-Interface sehr übersichtlich und leicht zu implementieren (siehe Quellcodeausschnitt im Anhang A.4). Im Vergleich zur SRAM-Kommunikation ist keine Adressberechnung notwendig, da die Daten der Reihenfolge nach abgearbeitet werden. Mit dem optional verwendbaren *Stream-In-Complete*-Signal können zudem ohne die Übergabe zusätzlicher Parameter aus Sicht der VHDL-Implementierung beliebig lange Streams verarbeitet werden. Während die eigentliche Operation (zwei 16-Bit-Additionen) sehr kurz beschrieben werden kann, müssen zusätzlich noch die Kontrollsignale zu den Core Services angesteuert und eine entsprechende Beschreibung der Schnittstelle (nicht im Quellcode-Listing enthalten) vorgenommen werden. Sobald ein Stream bereit ist, kann durch das Anfordern von Werten aus einem Stream gelesen werden. Einem Stream werden Daten hinzugefügt, indem ein valid-Signal für einen anliegenden Wert gesendet wird. Das Senden des Streams wird letztendlich durch das flush-Signal initiiert.

5.2.2 Tests und Messungen

Für eine erfolgreiche Umsetzung in Hardware musste trotz einer geringen Logiktiefe des Algortihmus für beide Implementierungen ein hoher Hardware-Synthese-Aufwand gewählt werden. Bei normalem Aufwand gelang es bei der Verdrahtung nicht, die zeitlichen Anforderungen (engl. Timing-Constraints) der Core Services zu erfüllen. Das gesamte Design benötigt bei gleichen Aufrufparametern der tech-

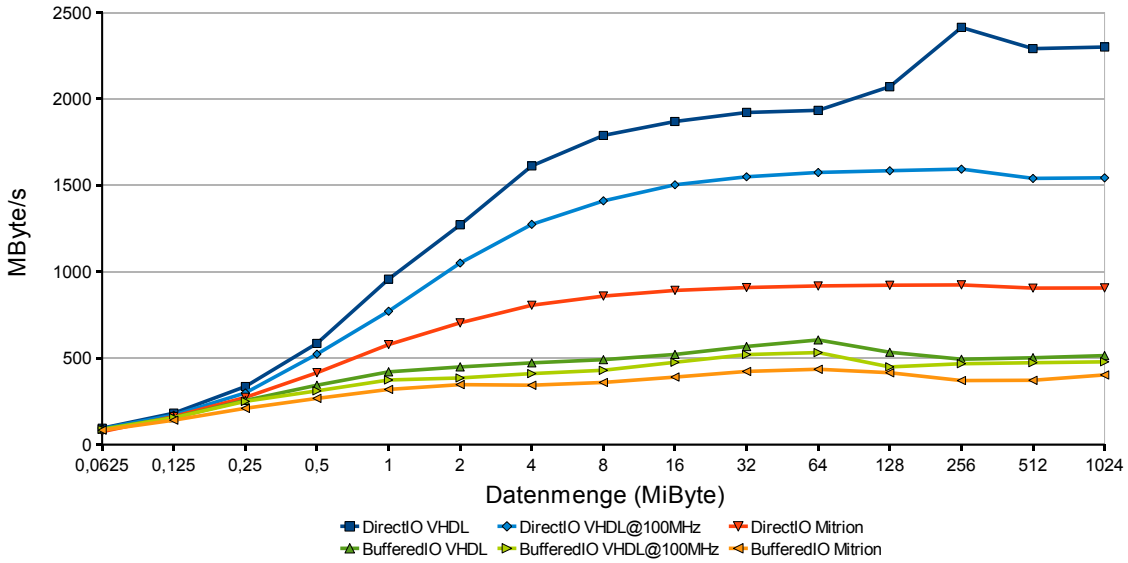


Abbildung 5.3: Direct Streaming Durchsatz

nologiespezifischen Synthese mit Mitrion 13% und mit der VHDL-Implementierung 7% der Slices des Virtex-4-LX200-FPGAs. Der deutlich größere Flächenbedarf des Mitrion-Designs begründet sich in der unnötigen Integration des Speicherkontrollers.

Während der Tests erreichte diese Kommunikationsvariante mit bis zu 2,4 GByte/s bei der Übertragung einer kompletten Hugepage die höchsten Datenraten. Abbildung 5.3 zeigt die Ergebnisse der durchgeführten Tests. Wie bei der SRAM-Kommunikation erhöht sich der Durchsatz mit der Größe der übertragenen Daten. Bei 256 MiByte wird für Direct I/O der maximale Durchsatz erreicht, welcher sich bei größeren Datenmengen leicht verringert. Dies liegt daran, dass mit einer Hugepage maximal 256 MiBytes zusammenhängender Speicher allokiert werden kann und die *rasclib_cop_send*-Funktion nur einen Pointer auf den Speicherbereich der Daten akzeptiert. Nach einem Streaming-Durchlauf muss dann die *rasclib_cop_close*-Funktion (siehe Tabelle 3.2) aufgerufen werden, bevor ein neuer Durchlauf gestartet werden kann. Größere Datenmengen wurden durch das Senden einer Hugepage in mehreren Durchläufen übertragen. Die Messschleife zur Ermittlung des Durchsatzes ist im Quellcode-Listing A.5 dargestellt.

Das Diagramm zeigt deutlich geringere Datenraten für Direct-Streaming mit Mitrion und Direct I/O. Während die VHDL-Implementierung mit 100 MHz Taktfrequenz sehr nahe an den theoretischen Maximalwert (von 1,6 GByte/s) herankommt, bleibt die Mitrion-Umsetzung unter einem GByte/s. Für Buffered I/O liegen die gemessenen Werte sehr nahe beieinander, wobei die VHDL-Implementierung mit 200 MHz die höchsten und die Mitrion-Implementierung die niedrigsten Ergebnisse liefern. Zudem schwanken die Werte zwischen vergleichbaren Messungen bei größeren Datenpaketen stärker als bei Direct I/O, weil eine Pufferung über einen Speicherbereich des Kernels hinzukommt und dieser mitunter nicht kontinuierlich angelegt werden kann.

5.3 Kommunikation über Memory Mapped Registers

Anwendungen mit geringer Host-FPGA-Kommunikation können die sog. Memory Mapped Registers (MMRs) nutzen. Mit Mitrion ist die Kommunikation über MMRs nur stark eingeschränkt möglich. Sie erlauben nur die Übergabe von Parametern und können als einfache Ausgabe von Ergebnissen oder Debug-Informationen verwendet werden. Eine Ping-Pong-Kommunikation, wie sie mit einer HDL möglich ist, kann mit Mitrion-C nicht vorgenommen werden. Während die Core Services Kontrollsignale bereitstellen um zu signalisieren, dass der Host ein Register verändert oder gelesen hat, ist diese Funktionalität in Mitrion-C nicht verfügbar, weshalb im Folgenden nur auf die VHDL-Implementierung und deren Messergebnisse eingegangen wird.

Der wesentliche Vorteil gegenüber den beiden bereits vorgestellten Kommunikationsvarianten liegt im wechselseitigen Datenaustausch zwischen Host und FPGA, ohne dass das Taktsignal des Algorithmus unterbrochen wird. Damit ist es dem Algorithmus während seiner Ausführung möglich, auf sich verändernde Eingaben zu reagieren. Auch der Host kann auf die Ausgaben des Algorithmus reagieren, ohne dessen Ausführung beenden zu müssen. Beim Direct Streaming und bei der SRAM-Kommunikation hingegen werden einmalig Daten geschickt und nach dem Start-Signal auf das Empfangen der Daten gewartet. Anschließend kann der Algorithmus mit anderen Daten neu gestartet werden. Die Kommunikation mit MMRs ist somit für den Langzeitbetrieb geeignet.

5.3.1 Implementierung

Über die RASC Core Services können dem Algorithmus-Block bis zu 64 sog. Algorithm Defined Registers (ADRs) als Modul-Eingänge zur Verfügung gestellt werden. Hinzu kommen drei Kontrollsignale pro Register, um dieses zu beschreiben oder aber zu erfahren, ob es vom Host beschrieben oder gelesen wurde. Bis zu 64 Debug-Register können zusätzlich genutzt werden, um während der Ausführung des Algorithmus auf dem FPGA eine Fehlersuche zu ermöglichen oder einfach nur Daten auszugeben.

Die Beispiel-Implementierung (vgl. VHDL-Quellcodeauschnitt A.6) stellt für Dateneingang und Datenausgang je 16 ADRs bereit. Um testen zu können, wie sich der Durchsatz mit den verwendeten ADRs verhält, kann die Anzahl dieser über einen vom Host bestimmbaren Parameter eingestellt werden. Ein Auschnitt aus der Hostanwendung ist in Listing A.7 aufgeführt. Da die ADRs nicht gleichzeitg von den Core Services beschrieben werden, wird immer der Wert eines gerade vom Host veränderten Registers in einem FIFO gepuffert. Ist dieser nicht leer, werden die enthaltenen Werte in die Ausgangsregister geschrieben. Enthalten alle aktuelle Werte, wird zusätzlich einem Debug-Register diese Information mitgeteilt, welche der Host durch Polling (periodische Anfragen) abrufen kann. Damit kann sich sowohl

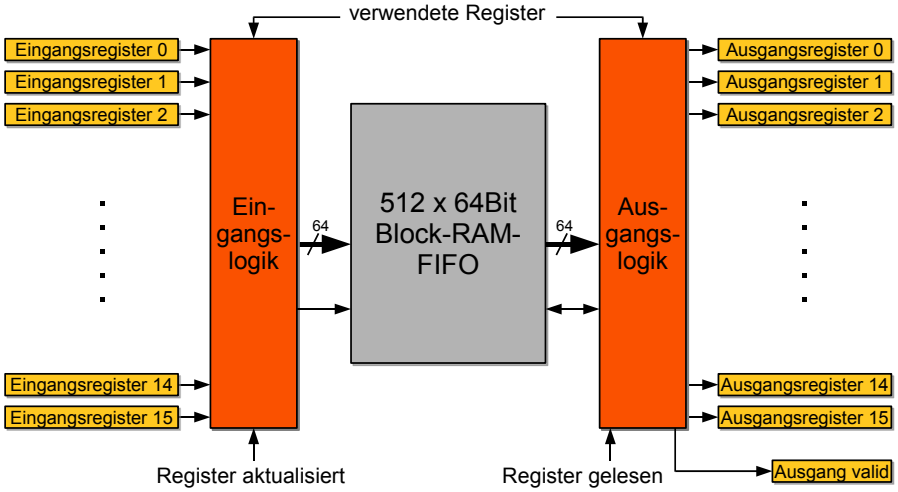


Abbildung 5.4: Memory Mapped Registers – Kommunikationsschema

der Algorithmus als auch der Host über den Zustand der Register informieren. Das Prinzip der Beispiel-Implementierung ist schematisch in Abbildung 5.4 dargestellt.

In diesem Entwurf wird ein mit dem Xilinx Core Generator erstellter, 512 Werte tiefer, 64-Bit First-Word-Fall-Through[2] Block-RAM-FIFO als Puffer genutzt. Zudem werden Zähler verwendet um die Latenz zwischen Register-Aktualisierungen und Register-Auslesevorgängen zu messen. Mit dem Host-Programm kann diese dann über Debug-Register ausgelesen werden.

Um weitere Ressourcen einzusparen, wäre es möglich die selben Register als Ein- und Ausgang zu verwenden. Allerdings muss dann dafür gesorgt werden, dass der Host und der Algorithmus nicht gleichzeitig auf diese Register schreiben können.

5.3.2 Tests und Messungen

Das Design benötigt laut Ausgaben der Hardware-Synthese 9% der Virtex-4-Slices und damit etwas mehr Ressourcen als die Direct Streaming Beispiel-Implementierung. Allerdings wurden mit 32 ADRs (aus Timing-Gründen zusätzlich gepuffert), acht Debug-Registern und einem 512 Werte tiefen 64-Bit-Block-RAM-FIFO schon vom Algorithmus mehr Ressourcen verwendet.

Die Durchsatzmessungen zeigen deutlich geringere Ergebnisse als beim Streaming. Abbildung 5.5 zeigt den erreichbaren Datendurchsatz bei der Übertragung von $512 * 64Bit = 4KiByte$ in Abhängigkeit von den verwendeten ADRs. Das hostseitige Polling ist in dieser Messung optional. Wird angenommen, dass die Ausgangsregister beim Auslesen immer gültig sind, ist keine Anfrage über deren Zustand notwendig und der Durchsatz erhöht sich. Eine solche Annahme kann daraus resultieren, dass zwischen dem Schrei-

[2] erster eingehender Wert wird durchgereicht und liegt direkt am Ausgang an

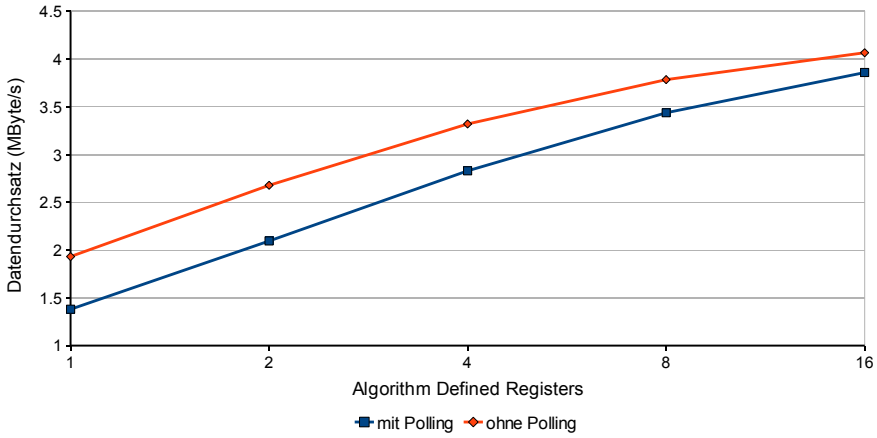


Abbildung 5.5: Memory Mapped Registers – Durchsatzmessung

ben und Lesen eines Registers durch den Host etwa 400 Takte bei $200MHz$ Taktfrequenz ($2\mu s$) liegen. Werden mehrere Pakete hintereinander vom Host gesendet oder empfangen, liegt die Latenz zwischen dem Eintreffen dieser bei ca. 180 Takten ($0,9\mu s$). Die Kontrolle, ob aktuelle Daten vorliegen, erfordert mindestens einen zusätzlichen Lesevorgang durch den Host, was den effektiven Datendurchsatz verringert.

Es müssen mindestens soviele 8-Byte-Pakete übertragen werden, wie Eingangs- bzw. Ausgangsregister genutzt werden. Wird ein ADR bei der Übertragung von nur acht Byte (64 Bit) pro Durchlauf verwendet, liegt der Datendurchsatz bei 0,67 MByte/s. Die Nutzung von 16 ADRs und 512 übertragenen Werten (4096 Bytes) pro Durchlauf erhöht den Durchsatz auf bis zu 4,07 MByte/s ohne Synchronisation beim Auslesen der Register. Die zusätzliche Abfrage, ob Registerwerte bereitstehen, verringert den Datendurchsatz je nach Anzahl der verwendeten ADRs um 0,2 bis 0,6 MByte/s. Die Verwendung von Direct I/O oder Buffered I/O hat bei diesen geringen Datenmengen jedoch keinen erkennbaren Einfluss auf Durchsatz oder Latenz.

5.4 MD5 Brute Force

Als Repräsentant für Kryptographische Verfahren wird in dieser Arbeit die Hashfunktion MD5 (Message-Digest Algorithm 5) verwendet. Sie erzeugt aus einem beliebig langen Klartext einen 128-Bit-Hashwert und wurde 1991 von Ronald Linn Rivest entwickelt. Als Internet Standard (RFC 1321, vgl. [Riv92]) wurde MD5 in einer Vielzahl von Anwendungen zur Verschlüsselung und wird immer noch zur Integritätsprüfung von Dateien verwendet. Bei letzterem wird die aktuell errechnete MD5-Summe einer Datei mit einer zuvor errechneten verglichen, um zu überprüfen, ob die Datei verändert oder beschädigt wurde. Bei der Übertragung von Dateien über das Internet oder auch in Netzwerken ist es üblich, eine MD5-

Hash-Summe zur Kontrolle der Übertragung mitzuschicken. Da für MD5 bekannt ist wie Kollisionen erzeugt werden können (vgl. [WY05]), wird es heutzutage immer weniger in Anwendungen wie SSL-Zertifikaten oder digitalen Signaturen verwendet, welche auf der Resistenz vor Kollisionen aufbauen.

Wird MD5 genutzt um einfache Passwörter zu speichern, so kann dieses mit sog. Regenbogentabellen (Sammlung von Klartextwörter und zugehörigem Hash) angegriffen werden. Das Hinzufügen eines Zufallswertes (Salt) zum Passwort verhindert solche Angriffe. Hierbei muss der Zufallswert mit dem Hash abgespeichert werden, um die Eindeutigkeit der Abbildung des Eingabewerts auf den Hash-Wert zu gewährleisten. Solche „salted Hashes“ können allerdings noch durch Ausprobieren aller Möglichkeiten (Brute Force) angegriffen und damit der Klartext ermittelt werden. Wieviele Zeichen ein Klartextwort haben muss, um sicher vor Brute-Force-Angriffen zu sein, zeigen die Performance-Messungen der vorgenommenen Implementierungen und Vergleiche aus anderen Arbeiten (siehe Abschnitt 5.4.3).

Ein MD5 Hash wird typischerweise als eine 32-stellige Hexadezimalzahl dargestellt. Folgendes Beispiel zeigt eine acht Zeichen/Byte lange ASCII-Eingabe und den zugehörigen MD5-Hash:

```
md5("rdietric") = d6fe591a6f217dc9c7dfd2766bd8b01d
```

Im Folgenden wird der Algorithmus zur Erzeugung eines MD5-Hashes aus einer Folge von ASCII-Zeichen erläutert, die VHDL- und Mitrion-C-Implementierung betrachtet und schließlich die Performance der verschiedenen Umsetzungen verglichen. Beide Implementierungen basieren auf der in C geschriebenen Kryptografie- und SSL-Bibliothek xyssl ([Dev07]), deren Quellcode frei zugänglich ist.

Algorithmus

Der MD5 Algorithmus wurde so entworfen, dass er schnell auf 32-Bit-Maschinen ausgeführt werden kann. Zudem benötigt er keine langen Substitutionstabellen und kann sehr kompakt programmiert werden.

MD5 erzeugt aus einer Nachricht mit variabler Länge eine Ausgabe fester Länge (128 Bit). Vorerst wird die Ausgangsnachricht so erweitert, dass ihre Länge (in Bits) kongruent zu 448 modulo 512 ist, indem eine Eins und dahinter Nullen angehängt werden. Besitzt die Nachricht bereits die geforderte Länge, wird dennoch eine Eins angehängt. Eine 64-Bit-Zahl enthält die Nachrichtenlänge und komplettiert die Nachricht, so dass sie durch 512 teilbar ist und blockweise als Eingabe verwendet werden kann.

Der Hauptalgorithmus von MD5 arbeitet mit einem Puffer aus vier 32-Bit-Wörtern, die zu Beginn mit festgelegten Konstanten initialisiert werden. Die Verarbeitung eines 512 Bit Nachrichtenblockes wird in vier sog. Runden vorgenommen. Diese sind einander ähnlich und bestehen aus jeweils 16 gleichen Operationen, basierend auf einer nichtlinearen Funktion, modularer Addition und einer Linkrotation.

Die in den vier Runden verwendeten Funktionen sind:

$$\begin{aligned} F(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ G(X,Y,Z) &= (X \wedge Z) \vee (Y \wedge \neg Z) \\ H(X,Y,Z) &= X \oplus Y \oplus Z \\ I(X,Y,Z) &= Y \oplus (X \vee \neg Z) \end{aligned}$$

Hierbei stehen $\oplus, \wedge, \vee, \neg$ für XOR, AND, OR und NOT-Operationen. Jeder Durchlauf arbeitet auf einem 512 Bit Nachrichtenblock und verändert den Puffer und damit die MD5-Summe. Werden nicht mehr als 56 Zeichen (ein Nachrichtenblock) verwendet, ist nur ein Durchlauf notwendig und die Schleife über die 512-Bit-Blöcke entfällt. Für die VHDL- und Mitrion-C-Implementierung werden ausschließlich kurze Nachrichten, von bis zu acht Zeichen betrachtet.

5.4.1 VHDL-Entwurf

Die Hardware-Beschreibung des MD5-Brute-Force-Algorithmus basiert auf mehreren parallel arbeitenden Pipelines, welche den Hauptteil des Algorithmus umsetzen. Die Parallelverarbeitung steckt damit zum einen in der Pipeline-Struktur selber und zum anderen in deren Anzahl. Hinzu kommt eine Komponente zur Erzeugung der 512-Bit-Nachrichtenblöcke und eine weitere zur Rückgewinnung des Klartextes, wenn der gesuchte Hash gefunden wurde. Eine schematische Übersicht der VHDL-Implementierung ist in Abbildung 5.6 dargestellt.

Die Erzeugung der Nachrichten wird im Klartext-Generator vorgenommen. Dieser basiert auf einer der Klartextlänge entsprechenden Anzahl von Zählern. Jeder dieser Zähler repräsentiert ein ASCII-Zeichen, wobei der zu durchsuchende Bereich der ASCII-Tabelle generisch gewählt werden kann. Der erste Zähler arbeitet mit einer der Pipelinezahl entsprechenden Schrittweite, wodurch ebenso viele Nachrichtenblöcke pro Takt erzeugt werden können. In diesem VHDL-Entwurf wird die Klartextlänge generisch festgelegt, was zum einen die Implementierung vereinfacht und zum anderen weniger FPGA-Ressourcen benötigt (durch Verwendung von weniger als 56 Zeichen).

Bei der Umsetzung der MD5-Pipeline besteht das wesentliche Problem in der effizienten und ressourcensparenden Bereitstellung der Daten in den einzelnen Stufen. Ein naiver Ansatz speichert eine Nachricht bis deren Verarbeitung erfolgt ist. Die erzeugten Zwischenergebnisse einer Stufe werden dabei temporär in Registern gehalten, bis sie nicht mehr benötigt werden. Eine Pipeline speichert damit immer soviele Nachrichten wie sie gerade verarbeitet ($Tiefe\ der\ Pipeline \cdot 512Bit$). Der Vorteil dieses Ansatzes liegt darin, dass der Klartext zu einem gefundenen Hash direkt anliegt. Nachteilig ist der immense Bedarf an

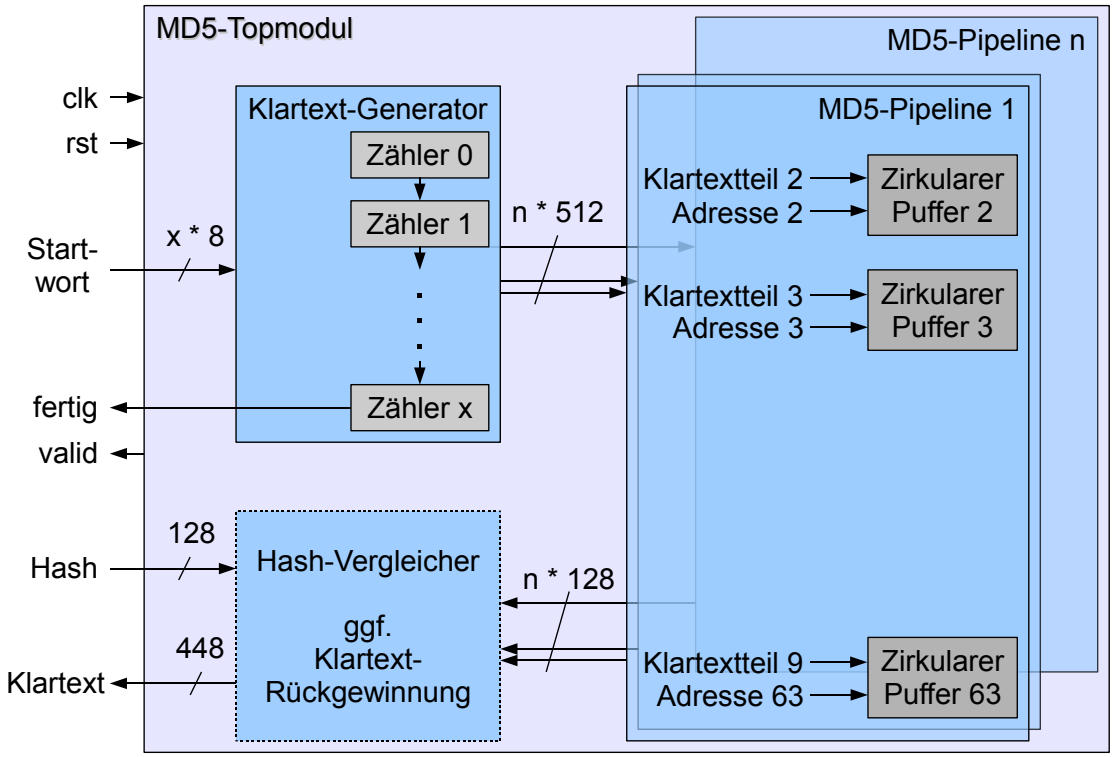


Abbildung 5.6: MD5 Brute Force – VHDL-Entwurf

FPGA-Fläche (allein 4096 Virtex-4-Slices für die Nachrichtenspeicherung einer Pipeline mit 128 Stufen, wenn pro Slice 16 Bit gespeichert werden).

Um den Flächenbedarf für die Speicherung der Nachrichtenblöcke zu verringern, kann der Block-RAM des FPGA genutzt werden. Zudem müssen Teile (32-Bit-Blöcke) der Nachricht nur solange gespeichert werden, bis sie nicht mehr benötigt werden, mindestens jedoch bis zur ersten Stufe der vierten Runde. Wenn vom allgemeinen Fall, dass 56 Zeichen pro Nachricht genutzt werden können und die Länge der Nachricht variiert, abgesehen wird, kann ein Großteil des Nachrichtenblockes als konstant angenommen werden. Die vorgenommene Implementierung passt sich der Wortlänge an und nutzt bei der Verwendung von vielen Zeichen den Block-RAM als zirkularen Puffer. Dieser benötigt nur einen Zähler, welcher die Schreib- und Leseadresse definiert. Damit können zwei Puffer in einen Dual-Port-Block-RAM integriert und Block-RAM-Ressourcen eingespart werden. Die VHDL-Funktionen der ersten beiden MD5-Runden, welche für die Umsetzung der Pipeline-Stufen verwendet wurden, sind in 5.3 aufgelistet.

Wie bei der Mitrion-C-Implementierung wird jede Stufe zum besseren Timing in zwei Operationen unterteilt, wodurch eine Pipeline mit einer Tiefe von 128 Stufen entsteht. Die Aufteilung der Einzeloperationen in jeder Stufe wurde im Vergleich zur Mitrion-C-Implementierung anders gewählt, um mit dem Design eine höhere Taktfrequenz erreichen zu können. In beiden Teilstufen werden jeweils zwei 32-Bit-Additionen vorgenommen. Die *md5_logic()*-Funktionen führen zusätzlich weitere Logik-Operationen

```
function md5_logic12(din0 : std_logic_vector(31 downto 0);
                     din1 : std_logic_vector(31 downto 0);
                     din2 : std_logic_vector(31 downto 0);
                     din3 : std_logic_vector(31 downto 0);
                     dinx : std_logic_vector(31 downto 0)
                    ) return std_logic_vector is
begin
  return din0 + (din1 XOR (din2 AND (din3 XOR din1))) + dinx;
end md5_logic12;

function md5_shift(tmp   : std_logic_vector(31 downto 0);
                   shift : positive;
                   din   : std_logic_vector(31 downto 0);
                   const : std_logic_vector(31 downto 0)
                  ) return std_logic_vector is
  variable to_shift : std_logic_vector(31 downto 0);
begin
  to_shift := tmp + const;
  return (std_logic_vector((unsigned(to_shift)) sll shift)
          OR std_logic_vector((unsigned(to_shift)) srl (32-shift))) + din;
end md5_shift;
```

Auflistung 5.3: Umsetzung der MD5-Operationen (Runden 1 und 2) in VHDL

durch, während die *md5_shift()*-Funktion eine Linksrotation vornimmt. Die Übergabeparameter `shift` und `const` sind Konstanten. Der Aufruf dieser Funktionen erfolgt genauso wie in anderen Programmiersprachen durch Übergabe der Parameter/Signalen. Ein Auschnitt der VHDL-MD5-Pipeline ist in A.8 gelistet.

Jeder von einer MD5-Pipeline erzeugte Hash wird mit dem gesuchten Hash verglichen und bei Übereinstimmung der zugehörige Klartext erzeugt. Der erzeugte Hash kommt in vier 32-Bit-Blöcken jeweils um zwei Takte verzögert aus der Pipeline, wodurch sich ein Vergleich der einzelnen Blöcke mit den zugehörigen Teilen des gesuchten Hashes anbietet (vgl. Quellcode A.9). Die Klartextrückgewinnung verwendet die Pipeline-Tiefe, die aktuellen Zähler des Klartextgenerators und die Nummer der Pipeline, welche den gesuchten Hash erzeugt hat. Pro reproduziertem Zeichen wird ein Takt benötigt. Ein entsprechender Quellcodeausschnitt ist in A.10 gelistet.

Das Design nutzt in der aktuellen Variante noch die Datentypen `std_logic` und `std_logic_vector`, welche von den meisten Synthese-Tools auch verarbeitet werden können. An dieser Stelle würde sich einen Anpassung mit dem Paket *numeric.std.all* anbieten. Die Nutzung von selbst geschriebenen Paketen und darin enthalten Funktionen und Prozeduren ermöglicht eine sehr kompakte und übersichtliche Beschreibung der Hardware.

Die Schnittstelle zu den Core Services kann einfach gehalten werden, da nur die Eingabe des Hashes (128 Bit) und die Ausgabe des Klartextes (maximal 56 *ASCII-Zeichen* $\cdot 8Bit = 448Bit$). Dementsprechend werden insgesamt neun ADRs (je 64 Bit) für die Datenübertragung verwendet. Der Algorithmus beendet sich, wenn der letzte Zähler übergelaufen ist (kein Klartext gefunden wurde) oder der erste

übereinstimmende Hash mit zugehörigem Klartext berechnet wurde. Über ein Debug-Register kann das Host-Programm ermitteln, in welchem Zustand (fertig, rechnend, Klartext gefunden) sich der Algorithmus befindet. Der C-Quellcode zur Kommunikation und Performance-Messung ist in A.11 gelistet.

Die derzeitige Hardware-Synthese kann acht MD5-Pipelines bei einer Taktfrequenz von 146 MHz umsetzen. Dabei werden Klartexte mit nur acht Buchstaben betrachtet. Ab einer Anzahl von neun Pipelines bricht das Synthese-Tool Xilinx ISE während der High-Level-Synthese ohne Angabe eines Design-Fehlers ab. Die Tabelle 5.1 zeigt die Nutzung der FPGA-Ressourcen laut den Angaben der Hardware-Synthese nach der Plazierung. Es kann also vermutet werden, dass noch mindestens eine Pipeline bei funktionierender High-Level-Synthese implementierbar ist.

Anzahl der verwendeten Slices	67870	von	89088	76%
Anzahl der Slice Flipflops	104808	von	178176	58%
Anzahl der 4-Eingangs-LUTs	118693	von	178176	66%
Anzahl der FIFO16/RAMB16s	58	von	336	17%

Tabelle 5.1: MD5 Brute-Force: FPGA Ressourcen-Nutzung – VHDL

5.4.2 Implementierung mit Mitrion-C

Die Implementierung eines MD5-Brute-Force-Algorithmus in Mitrion-C wurde bereits im Zusammenhang mit dem Technischen Report [IJ08] vorgenommen und lehnt sich insbesondere bei der Beschreibung der MD5-Pipeline stark an den C-Code der xyssl-Bibliothek an. In 5.4 sind der Zustandspuffer der MD5-Summe und einige Stufen der Pipeline aufgelistet. Die angegebenen Konstanten sind vom Algorithmus vorgegeben. Das Kernstück des Algorithmus ist somit leicht in Mitrion-C zu übertragen, zumal die Datenabhängigkeiten zwischen den einzelnen Zuweisungen automatisch gehandhabt werden – in VHDL muss der Programmierer für deren Einhaltung sorgen. Wie im Quellcode-Listing zu sehen ist, werden die Operationen pro Stufe auf zwei Zuweisungen aufgeteilt. Vor der Umsetzung in Hardware, führt der Mitrion-Compiler Optimierungen durch und passt die Abfolge der Operationen, wie sie im Quellcode stehen, an. Das Ergebnis ist eine 128-stufige Pipeline. Um diese mit Daten zu versorgen, wurde eine Verschachtelung von FOREACH-Schleifen (entsprechend der Wortlänge) über einen Bereich der ASCII-Tabelle (z.B. Kleinbuchstaben) verwendet. Das komplettieren des Nachrichtenblocks kann dann ähnlich wie in der VHDL-Implementierung durch die feste Zeichenlänge einfach gehalten werden.

Zudem kann aus dem Datenabhängigkeitsgraphen des Mitrion-Simulators abgelesen werden, dass für die Speicherung des Klartext-Nachrichtenblockes wie in der VHDL-Implementierung Block-RAM verwendet wird. Der Mitrion-Compiler verwendet zur Speicherung von Vektoren ab einer gewissen Gesamt-Bitbreite prinzipiell Block-RAM.

```
uint:32 a_0 = state_in[0];
uint:32 b_0 = state_in[1];
uint:32 c_0 = state_in[2];
uint:32 d_0 = state_in[3];

// erste Runde, Stufe 1 und 2
uint:32 a_1 = a_0 + (d_0 ^ (b_0 & (c_0 ^ d_0))) + x[0] + 0xD76AA478;
uint:32 a_2 = ((a_1 << 7) | (a_1 >> (32 - 7))) + b_0;
uint:32 d_1 = d_0 + (c_0 ^ (a_2 & (b_0 ^ c_0))) + x[1] + 0xE8C7B756;
uint:32 d_2 = ((d_1 << 12) | (d_1 >> (32 - 12))) + a_2;

// zweite Runde, erste Stufe
uint:32 a_9 = a_8 + (c_8 ^ (d_8 & (b_8 ^ c_8))) + x[1] + 0xF61E2562;
uint:32 a_10 = ((a_9 << 5) | (a_9 >> (32 - 5))) + b_8;

// dritte Runde, erste Stufe
uint:32 a_17 = a_16 + (b_16 ^ c_16 ^ d_16) + x[5] + 0xFFFA3942;
uint:32 a_18 = ((a_17 << 4) | (a_17 >> (32 - 4))) + b_16;

// vierte Runde, letzte Stufe
uint:32 b_31 = b_30 + (d_32 ^ (c_32 | ~a_32 )) + x[9] + 0xEB86D391;
uint:32 b_32 = ((b_31 << 21) | (b_31 >> (32 - 21))) + c_32;

uint:32 s0_out = state_in[0] + a_32;
uint:32 s1_out = state_in[1] + b_32;
uint:32 s2_out = state_in[2] + c_32;
uint:32 s3_out = state_in[3] + d_32;
state_out = [s0_out, s1_out, s2_out, s3_out];
```

Auflistung 5.4: Mitrion-C MD5-Pipeline

Mit dem aus der Mitrion-C-Beschreibung generierten VHDL-Design lassen sich durch den Overhead der MVP-Verarbeitungseinheiten nur drei Pipelines auf dem Virtex-4-LX200-FPGA umsetzen, wobei für die Beispiel-Implementierung auch nur acht Zeichen betrachtet wurden. Die Tabelle 5.2 zeigt die Nutzung der FPGA-Ressourcen laut den Angaben der Hardware-Synthese nach der Plazierung. Die vor der Synthese vorgenommene Abschätzung durch Mitrion (von 65% Flipflop- und 48% Block-RAM-Nutzung) ist beim MD5-Algorithmus mit acht Zeichen und drei Pipelines sehr gut und stimmt fast mit den Ergebnissen der Hardware-Synthese überein.

Anzahl der verwendeten Slices	70131	von	89088	79%
Anzahl der Slice Flipflops	109008	von	178176	61%
Anzahl der 4-Eingangs-LUTs	116749	von	178176	65%
Anzahl der FIFO16/RAMB16s	164	von	336	48%

Tabelle 5.2: MD5 Brute-Force: FPGA Ressourcen-Nutzung – Mitrion

Obwohl die Ergebnisse vermuten lassen, dass noch eine weitere Pipeline implementierbar ist, konnte die Verdrahtung des entsprechenden Designs wegen Timing-Fehlern nicht erfolgreich beendet werden. Auch durch Anpassung der Synthese-Parameter konnten keine vier Pipelines implementiert werden. Hinzu kommt, dass ein Ergebnis erst nach der Durchsuchung des gesamten Zeichenraums (aller Zeichenkombinationen) zur Verfügung steht, wodurch sich die durchschnittliche Suchzeit verdoppelt. Ein bis zur

Version 2.0.1 nicht vorhandenes Mitrion-C-Sprachkonstrukt zur vorzeitigen Beendung des Algorithmus würde dieses Problem beseitigen.

5.4.3 Vergleich der MD5-Implementierungen

Die für die RASC-Plattform vorgenommenen Implementierungen verwenden beide in etwa gleich viele Logik-Ressourcen, wobei das Mitrion-C-Design dreimal soviele Block-RAM-Ressourcen benötigt. In beiden Fällen sind noch über 20% der FPGA-Fläche ungenutzt. Während mit der aktuellen Version von Mitrion keine weitere MD5-Pipeline in das Design integriert werden kann, sollte es für den VHDL-Entwurf (durch anderes Synthese-Tool, Änderung im Entwurf) möglich sein. Tabelle 5.3 zeigt die Performance beider Implementierungen.

	Mitrion-C@100MHz			VHDL@146,6MHz	
parallele Pipelines	1	2	3	3	8
M Hashes/s (theoretisch)	100	200	300	440	1173
M Hashes/s (gemessen)	100	200	288	440	1173
Effizienz	100%	100%	96%	100%	100%

Tabelle 5.3: Leistungsvergleich der MD5-Hardware-Implementierungen

Bei der Mitrion-Umsetzung ist ein leichter Rückgang der Effizienz mit drei Pipelines zu erkennen. Dies könnte daran liegen, dass die Generierung des Klartextes aus Timing-Gründen mit mehr Pipelines verändert werden muss. Eine exakte Aussage darüber kann jedoch aufgrund des compiler-generierten Designs nicht getroffen werden. Die Effizienz der VHDL-Umsetzungen liegt konstant bei rund 100%. Auch die Generierung und Rückgewinnung des Klartextes bleibt hinsichtlich ihrer Ressourcen-Ausnutzung nahezu unabhängig von der Anzahl der Pipelines.

Ein Vergleich der vorgenommenen Hardware-Implementierungen mit Software-Umsetzungen auf Grafikkarten und CPUs zeigt Abbildung 5.7. Die Messungergebnisse der Nvidia-Grafikprozessoren wurden [AD09] entnommen, während die MD5-Brute-Force-Performance der Prozessoren AthlonX2 5000+ und Core2Duo E8400 und des Grafikprozessors Radeon 3850 mit BarsWF MD5 BruteForcer v0.8 auf normalen Desktop-Rechnern gemessen wurde. Der Test des Intel Itanium 2 Prozessors basiert auf der xyssl-Bibliothek, welche auch als Grundlage für die Hardware-Implementierungen verwendet wurde und keine Optimierungen hinsichtlich Befehlsatzerweiterung moderner Prozessoren nutzt. BarsWF 0.8 hingegen gilt derzeit als die schnellste Software-Lösung zur Brute-Force-Suche von MD5-Hashes und unterstützt ausschließlich Prozessoren mit der SSE2-Erweiterung. Zudem liegt es auf CUDA oder Brook basierend zur Nutzung von Grafikkarten vor.

Werden 67 verschiedene Zeichen (Groß-/Kleinbuchstaben, Ziffern, fünf Sonderzeichen) und eine Wort-

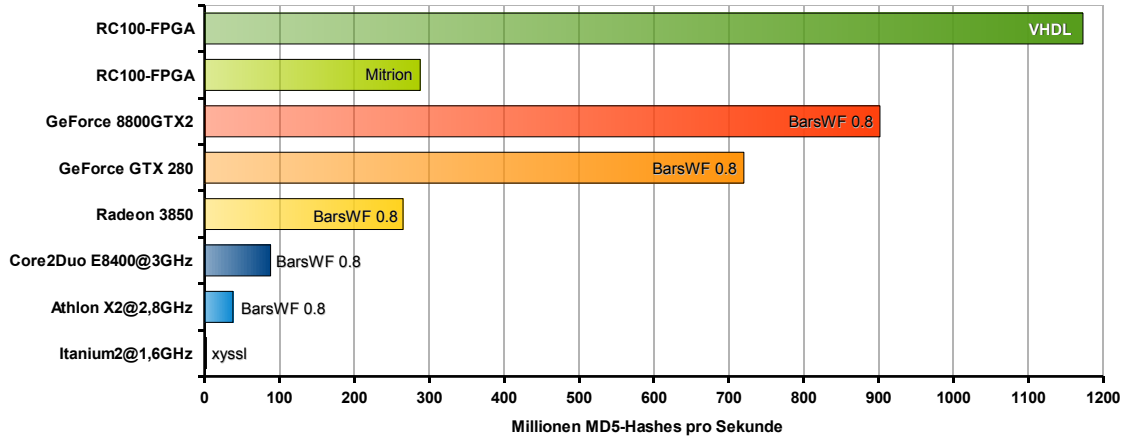


Abbildung 5.7: MD5 Brute Force – Performance Vergleich

länge von acht Zeichen betrachtet, können daraus 67^8 verschiedene Wörter gebildet werden. Die schnellste hier vorgestellte Implementierung (VHDL-Entwurf auf RC100-FPGA) benötigt maximal 4,1 Tage um das Klartextwort zu so einem MD5-Hash zu finden. Bei einer Länge von zehn Zeichen lässt es sich mit 64 FPGAs im Durchschnitt in etwa 21 Wochen ermitteln. Auf MD5 basierende Passwörter aus zwölf Zeichen sind dagegen nach aktuellem Kenntnisstand für hinreichend lange Zeit sicher vor solchen Angriffen. Tabelle 5.4 zeigt eine Übersicht der maximalen Suchdauer von Klartextworten verschiedener Länge zu MD5-Hashes durch die schnellste hier betrachtete Implementierung.

Wortlänge	Zeichenraum	maximale Suchdauer	
6	a-z, A-Z	17,4	Sekunden
6	95 ASCII-Zeichen	10,8	Minuten
8	a-z, A-Z	13,1	Stunden
8	a-z, A-Z, 0-9, 5 Sonderzeichen	4,1	Tage
8	95 ASCII-Zeichen	9,7	Wochen
10	a-z, A-Z	4,0	Jahre
10	95 ASCII-Zeichen	1673	Jahre

Tabelle 5.4: Maximale Suchdauer MD5-basierter Passwörter (VHDL-Implementierung)

5.5 Damenproblem

Beim klassischen Damenproblem sollen acht Damen auf einem typischen Schachbrett (8×8 Felder) so positioniert werden, dass sich keine zwei davon nach den Schachregeln in einem Zug schlagen können. Die Figurenfarbe ist dabei irrelevant und als Annahme gilt, dass jede Figur eine beliebige andere schlagen könnte. Um eine Lösung zu finden, dürfen demnach keine zwei Damen in der gleichen Reihe, Spalte oder Diagonale plaziert werden. Abbildung 5.8 zeigt eine Plazierung, die diesen Anforderungen genügt. Für das klassische Damenproblem gibt es 92 Lösungen. Wenn solche, die durch Spiegeln oder Rotieren des

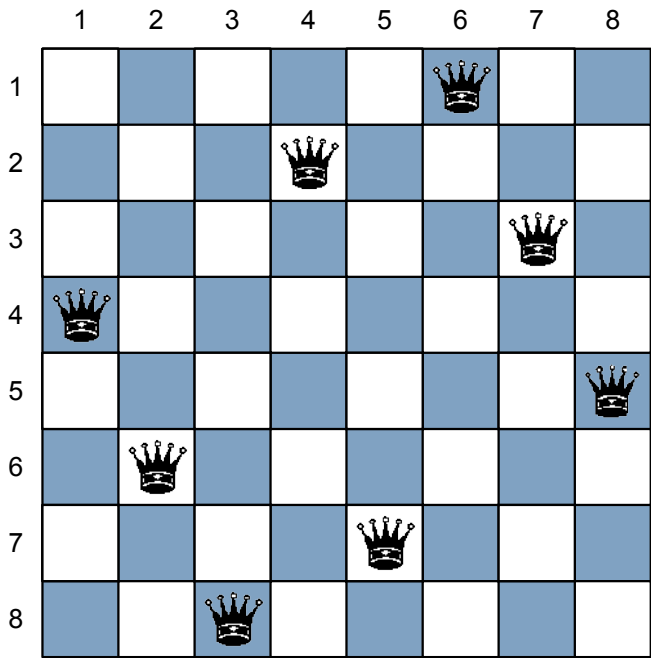


Abbildung 5.8: Eine Lösung des Acht-Damenproblems

Brettes aufeinander abgebildet werden können, nur einmal gezählt werden, verbleiben zwölf eindeutige Lösungen. Ausgehend von diesen könnte erwartet werden, dass durch vier möglichen Drehungswinkel und vier Spiegelungen $12 \cdot 8 = 96$ Lösungen existieren. Da jedoch einige Plazierungen selbstsymmetrisch sind und beim Drehen in sich selber übergehen, verringert sich die Lösungsmenge (vgl. [PNS] - Fakten).

Das Problem lässt sich auf die Plazierung von N Damen auf einem N × N Schachbrett verallgemeinern, wobei nur Lösungen für $N = 1$ und $N \geq 4$ existieren. Die eigentliche Problemstellung sucht nach der Anzahl der Lösungen für ein Damenproblem und damit nach allen möglichen Plazierungen, die den angegebenen Anforderungen genügen. Bis heute ist keine Formel zur Berechnung des N-Damenproblems bekannt. Als obere Schranke für die Lösungsanzahl gilt jedoch N Fakultät (N!), was die Anzahl für N einander nicht bedrohender Türme ist.

Aus dem Mangel an einer Berechnungsformel zur Ermittlung der Lösungen, kommen anstelle dessen Algorithmen zum Einsatz. Die meisten schnellen Implementierungen arbeiten mit dem sog. Backtracking-Verfahren. Dabei wird nach dem Versuch-und-Irrtum-Prinzip (engl. trial and error) vorgegangen, d.h. es wird versucht eine erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen. Kann eine Teillösung nicht zu einer Gesamtlösung führen, wird der letzte Schritt rückgängig gemacht und stattdessen ein alternativer Weg verfolgt. Es werden alle in Frage kommenden Lösungswege ausprobiert und damit auch alle Lösungen gefunden. Backtracking arbeitet damit nach dem Prinzip der Tiefensuche.

Im Folgenden sollen der Algorithmus allgemein und die VHDL- und Mitrion-C-Implementierung eines Backtracking-Algorithmus vorgestellt werden. An dieser Stelle soll weniger die exakte Umsetzung des

Algorithmus in Hardware/VHDL, als dessen Prinzip und die Schnittstelle zu RASC betrachtet werden. Anschließend wird die erreichte Performance der einzelnen Implementierungen verglichen.

5.5.1 Backtracking-Algorithmus und Parallelisierung

An dieser Stelle soll kurz erklärt werden, wie ein Algorithmus arbeitet, der alle Lösungen eines Damenproblems mit dem Backtracking-Verfahren findet. Es wird angenommen, dass die Tiefensuche spaltenweise vorgenommen wird. Das Nassi-Shneiderman-Diagramm in Abbildung 5.9 zeigt das Prinzip des Algorithmus. Hierbei muss beachtet werden, dass freie (durch keine Dame bedrohte) Felder in einer Spalte nur in eine Richtung (nach oben oder unten) besetzt werden können. Dies ist beim Rückspringen von Bedeutung, weil die dort bereits plaziert Dame nur auf freie Felder in der vereinbarten Richtung gesetzt werden darf. Beim Backtracking muss eine Dame entfernt werden, wenn zweimal hintereinander nicht gesetzt bzw. umgesetzt werden konnte.

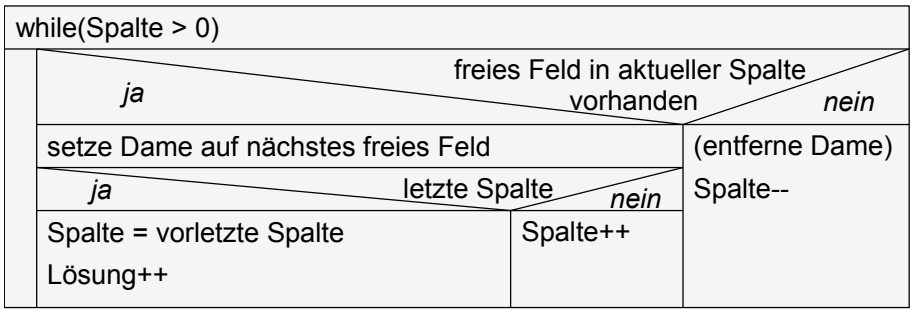


Abbildung 5.9: Damenproblem Backtracking-Algorithmus

Während ein naiver Brute-Force-Algorithmus für das klassische Damenproblem $64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57$ (fast $60, 5^8$) Plazierungen vornimmt und zusätzlich deren Gültigkeit überprüft, setzt eine effizientere Variante in jeder Spalte nur eine Dame (8^8 Plazierungen). Das Backtracking-Verfahren kann als sehr effizienter Brute-Force-Algorithmus angesehen werden und reduziert die durchgeführten Plazierungen noch einmal deutlich auf unter acht Fakultät ($8!$). Eine der heute noch schnellsten Software-Lösungen des N-Damenproblems wurde von Jeff Somers in der Programmiersprache C geschrieben (siehe [Som02]). Eine Übersicht der Lösungen einiger bisher berechneter Damenprobleme bietet Tabelle 5.5.

Eine Tiefensuche eignet sich hervorragend zur Parallelisierung, was bei den hier betrachteten Implementierungen mit Vorbelegungen von Spalten vorgenommen wurde. Dabei werden Damen bereits konfliktfrei in den ersten Spalten plaziert und das Problem in Teilbäume aufgegliedert. Mit steigender Anzahl vorplazierter Spalten steigt auch die Anzahl versteckter Konflikte, wodurch einige Teilbäume zu keiner Lösung mehr führen. Zudem erhöht sich der Berechnungsaufwand für die Vorplazierungen, während er für die Teilprobleme sinkt. Eine genauere Betrachtung dieses Problems wird in [PNS09] vorgenommen.

N	eindeutige Lösungen	alle Lösungen
1	1	1
5	1	10
8	12	92
15	2,85053E5	2,279184E6
19	6,21012754E8	4,968057848E9
22	3,36376244042E11	2,691008701644E12
23	3,029242658210E12	2,4233937684440E13
24	2,8439272956934E13	2,27514171973736E14
25	2,75986683743434E14	2,207893435808352E15
26	?	?

Tabelle 5.5: Lösungen einiger N-Damenprobleme

Das Queens@TUD-Projekt, und damit auch die hier betrachteten Implementierungen, verwenden sechs vorplazierte Spalten, um das 26-Damenproblem in Teilprobleme aufzuspalten. Dies hält den Kommunikationsaufwand gering und halbiert zudem noch die zu berechnenden Teilprobleme durch die sich an der Mitteldiagonalen spiegelnden, paarweise identischen Lösungen. Die sechs Damen der Vorplazierung können in 25.204.802 Konfigurationen aufgestellt werden, wodurch sich die daraus ergebenden Teilprobleme über Lösungsinstanzen (engl. Solver) unabhängig voneinander berechnen lassen.

5.5.2 VHDL-Implementierung

Das VHDL-Design basiert, wie alle schnellen Algorithmen zur Lösung des Damenproblems, auf dem Backtracking-Verfahren. Um möglichst feingranulare Parallelität zu erreichen und die Dauer der Lösung einzelner Teilprobleme gering zu halten, wurde eine an die Hardware (FPGA) angepasste Implementierung vorgenommen (siehe [PNS09]). In Software und auch bei der vorgenommenen Mitrion-C-Implementierung wird mit Feldern (engl. arrays) von Blockierungsvektoren (drei pro Spalte) gearbeitet, während sich für die Hardware-Umsetzung drei globale Vektoren für diagonal steigende, horizontale und diagonal fallende Blockierungen als effizienter erwiesen haben. Damit gilt immer nur eine Spalte als aktiv. Bei einem Rücksprung wird dann nicht die für diese Spalte bereits erzeugte Blockierung aus dem Speicher geladen, sondern durch Anpassung der globalen Blockierung erneut erzeugt. In Hardware ist das in einem Takt möglich und kann über Schieberegister mit geringem Ressourcenaufwand realisiert werden.

Während frühere Hardware-Solver (Komponent zur Berechnung eines Teilproblems) eine Dame pro Takt setzen und damit ein Teilproblem schneller berechnen konnten, wird in der aktuellen Implementierung in einem Takt entweder eine Spalte zurückgesprungen oder eine Dame gesetzt, um im folgenden Takt mit

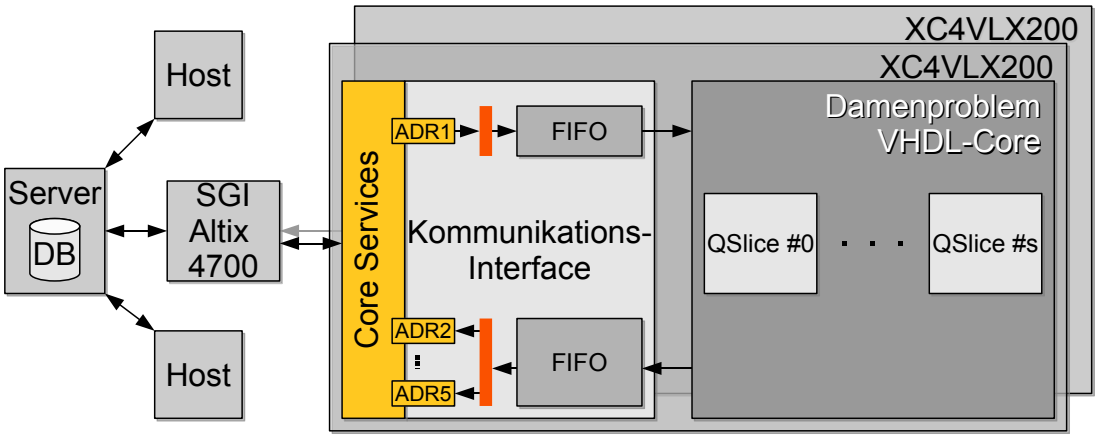


Abbildung 5.10: Infrastruktur und RASC-Anbindung von Queens@TUD

der nächsten Spalte fortzufahren. Diesen Geschwindigkeitsverlust gleichen die aktuellen Solver (oder Lösungsinstanzen) mit geringerem Ressourcenbedarf und einem kürzeren kritischen Pfad aus. Dadurch kann das Gesamt-Design mit höheren Taktfrequenzen betrieben werden und mehr Teilprobleme gleichzeitig berechnen. Die feinere Granularität der Solver ermöglicht zudem noch eine bessere Ressourcenausnutzung des FPGA.

Die wesentliche Aufgabe für diese Arbeit bestand in der Adaption des Damenproblem-Hardware-Designs an die SGI RASC Plattform. Abbildung 5.10 zeigt den prinzipiellen Aufbau und die Infrastruktur des Queens@TUD-Projektes. Die Kommunikationsschnittstelle und damit die RASC-Anbindung wurde hervorgehoben.

Es stehen zwei Xilinx Virtex-4-LX200-FPGAs (ein RC100-Blade, vgl. Abschnitt 3.2), welche in die SGI Altix 4700 (vgl. Abschnitt 3.1) integriert sind, zur Verfügung. Auf einem zentralen Server beinhaltet eine Datenbank alle ausstehenden und bereits berechneten Teilprobleme. Auf den Hosts läuft eine Java-Anwendung, welche noch nicht berechnete Vorplazierungen anfordert, die Kommunikation mit einem oder mehreren FPGAs durchführt, um schließlich gelöste Teilprobleme wieder an die Datenbank zu senden. Im Falle von RASC wird die Ansteuerung der FPGAs über das Java Native Interface (JNI) und die RASC C-Bibliothek (vgl. Abschnitt 3.4.1) vorgenommen. Der Quellcode der Damenproblem-Java-Schnittstelle für SGI RASC ist in A.16 gelistet, die zugehörigen nativen C-Funktionen in A.15.

Als RASC Kommunikationsvariante werden *Memory Mapped Registers* verwendet. Die Implementierung erfolgt ähnlich wie in Abschnitt 5.3.1 bereits erläutert. Es werden ein Eingangs- und vier Ausgangregister verwendet, um gleichzeitig zwei Teilprobleme empfangen und mit Lösung wieder zurückschicken zu können. Auf der Eingangsseite werden die empfangenen Werte jeweils in zwei Takten hintereinander in einen 32 Bit breiten und 256 Elemente tiefen Block-RAM-FIFO geschrieben, um bei Bedarf vom Damenproblem-Topmodul gelesen zu werden. Auf der Ausgangsseite übernimmt ein ebenso tiefer,

	Kommunikation	Damenproblem	Gesamtdesign	verfügbar
Slices	5835 (7%)	70370 (79%)	76205 (86%)	89088
FFs	6930 (4%)	106217 (60%)	113147 (63%)	178176
LUTs	6896 (4%)	129880 (73%)	136776 (76%)	178176
BRAM	14 (4%)	0	14 (4%)	336

Tabelle 5.6: 26-Damenproblem: Ressourcen-Nutzung – VHDL

aber 96 Bit breiter Block-RAM-FIFO die Pufferung der Werte (Vorplazierung+Lösung). Eine minimale Ressourceneinsparung wäre noch durch exakte Anpassung der ADRs und FIFOs an die wirklich benötigten Bitbreiten (29 Bit für die Vorplazierung und 46 Bit für die Lösung) möglich. Durch die Verwendung von Block-RAM kosten die tiefen FIFOs kaum Ressourcen und können für den unwahrscheinlichen Fall, dass alle Teilprobleme mit der Berechnung gleichzeitig fertig werden, die Lösungen puffern und neue Vorplazierungen bereitstellen. Die Blöcke zwischen den FIFOs und ADRs stellen die Empfangs- und Sendelogik dar. Insgesamt bleibt der Ressourcenbedarf des Kommunikationsmoduls unter 1% der FPGA-Fläche, wobei für die gesamte Kommunikation noch die Ressourcen der Core Services hinzukommen. Eine Übersicht der Ressourcenausnutzung der kompletten Damenproblem-Implementierung bietet Tabelle 5.6.

Die Host-Anwendung initiiert durch periodische Anfragen (engl. Polling) die Lösungsübertragung. Dabei wird zur Entlastung der Kommunikation nach einer erfolglosen Anfrage 30 Sekunden gewartet. Für jede empfangene Lösung wird wieder eine Vorbelegung an den FPGA gesendet.

Weder das Direct Streaming noch die Kommunikation über den SRAM eignen sich zur Integration in die Queens@TUD-Infrastruktur, da sie keinen dauerhaften Betrieb mit zwischenzeitlicher Ergebnisabfrage erlauben. Daten werden hier pro Durchlauf einmal gesendet und empfangen. Zwischen den Durchläufen wird das Taktsignal des Algorithmus angehalten und das Reset-Signal aktiviert. Damit würden alle sich in Berechnung befindenden Teilprobleme abgebrochen. Hinzu kommt, dass Teilprobleme als verloren gelten, wenn ihre Berechnung zu lange dauert. Werden nun sehr viele Teilprobleme gleichzeitig an den FPGA gesendet, um die Solver gut auszulasten, werden auch alle Lösungen gleichzeitig zurückgeschickt. Benötigt diese Gesamtberechnung zuviel Zeit, gelten die Lösungen als verloren. Außerdem würden alle in einem Durchlauf bereits berechneten Teillösungen bei einem Systemabsturz verloren gehen. Werden hingegen weniger Teilprobleme pro Durchlauf verarbeitet, sinkt die Auslastung der Solver durch die Ein- und Auslaufphase.

Das Design kann bis zu 148 Teilprobleme gleichzeitig berechnen und arbeitet mit einer Taktfrequenz von $126,\overline{6}MHz$, was laut [PNS] $148 \cdot 126,\overline{6} \approx 187SE$ (Slice Äquivalenten/auf 100MHz Löserinstanzen) entspricht. Eine schnellere Taktung verhindert das als Baumstruktur realisierte Verteilungsnetzwerk, welches die FIFOs mit den Solvern verbindet. Die großen Virtex-4 FPGAs ermöglichen die Implemen-

tierung sehr vieler Solver, wodurch sich der kritische Pfad des Verteilungsnetzwerkes verlängert. Eine Verringerung der Taktfrequenz von ausschließlich dieser Komponente, könnte das Gesamt-Design noch einmal beschleunigen.

5.5.3 Implementierung mit Mitrion-C

Der Mitrion-C-Programmcode wurde von einer C-Implementierung abgeleitet, die dem Algorithmus von Jeff Somers sehr ähnlich ist. Eine erste funktionierende Formulierung des Algorithmus in Mitrion-C erforderte etwa 20 Stunden Arbeitszeit und brachte den in Anhang A.12 gelisteten Quellcode. Er beschreibt im Wesentlichen einen einfachen Backtracking-Algorithmus zur Lösung eines Teilproblems, welches in Form einer Vorplazierung übergeben wird. Daraus werden die Blockierungsvektoren und die noch zu testenden freien Felder der ersten betrachteten Spalte ermittelt (Zeilen 22 bis 49). Die *mem-create*-Anweisungen allokieren Block-RAM und ersetzen die Funktionalität von Arrays aus typischen Programmiersprachen. Die *tuple*-Anweisungen fasst mehrere Speichertoken zusammen, da in Mitrion-C nur ein Speichertoken aus einer Blockanweisung propagiert werden kann, für diese Implementierung allerdings mehrere benötigt werden. Die *untup*-Anweisung entpackt die Elemente wieder aus einem Tuple. Für das 26-Damenproblem betrachtet die WHILE-Schleife genau 20 Spalten. Sie beinhaltet zwei ineinander geschachtelte Bedingungen. Die Erste überprüft, ob eine Spalte noch freie Felder (keine Dame wurde zuvor versucht zu plazieren und nicht blockiert) enthält. Ist dies nicht der Fall, wird zurückgesprungen. Ansonsten wird mit der zweiten Bedingung überprüft, ob es sich um die letzte Spalte handelt und ggf. die Lösung gezählt oder aber die Blockierungen und die freien Felder angepasst. Der Mitrion-C-Bandbreitengraph für ein Teilproblem ist im Anhang A.13 zu finden und gibt die kritischen WHILE-Schleife mit einer Latenz von 15 Takten an.

Für die gleichzeitige Berechnung von 20 Test-Vorbelegungen benötigte die daraus entstandene Hardware-Umsetzung fast drei Stunden (VHDL-Umsetzung benötigte knapp 13 Minuten) und es konnten nur etwa 20 Teilprobleme auf einem FPGA gleichzeitig verarbeitet werden. Zu beachten ist, dass die Vorbelegung mit der längsten Berechnung die dominierende ist und damit die Gesamtdauer bestimmt.

Dieses enttäuschende Ergebnis, welches langsamer als jeder Software-Algorithmus war, führte zu drei wesentlichen hardwarespezifischen Optimierungen (vgl. Quellcode A.14). Schreib- und Leseoperationen der Blockierungsvektoren wurden zu einer Operation zusammengefasst, was zum einen Block-RAM-Ressourcen einsparen konnte und zum anderen die Latenz der Schleife verringerte. Desweiteren wurden die benötigten Bitbreiten mittels Parametern an die Problemgröße angepasst, was wiederum FPGA-Ressourcen einsparen konnte. Als zusätzliche Optimierung wurden die Anpassung der Blockierung und des nächsten freien Feldes in einer Spalte aus der innersten Bedingungsanweisung herausgezogen und in

```
0 //----------------------- Summary: Bandwidth Graph -------------------------
  ROOT
  |  l: 6.0 ch: 6 rch : 6  BW limited. body ch: 6 rch: 6
     |--1
     | iterations: 1
5    |   Env/main/foreach<1>
     |  l: 1.0 ch: 1 rch : 6  BW limited. body ch: 1 rch: 6
     |--2
     | iterations: 6
     |   Env/main/foreach[1]/solveQ26/for<6>
10   |  l: 12.0 ch: 6 rch : 6  Latency limited dataloop. body latency: 2.0
     |--3
     | iterations: 1
     |   Env/main/foreach[1]/solveQ26/while
     |  l: 11.0 ch: 1 rch : 6  Latency limited dataloop. body latency: 11.0
15 //---------------------- End Summary: Bandwidth Graph -----------------------
  51 MemRead 7: 0x0ad0202e00008751001  // Blockierung aktuelle Spalte lesen
  52 MemRead 7: 0x3002fdc              // noch zu testende Felder lesen
  56(11) Watch slot: 0x3002fdc         // Lesevorgang beendet
  57(11) Watch cslots: 0x3002fd8       // Felder der aktuellen Spalte anpassen
20 58 MemWrite 7: 0x3002fd8            // ... schreiben
  58 MemWrite 8: 0x0568125c00030751005 // Blockierung naechsten Spalte schreiben
  59(11) Watch slots_nc: 0x28a87e0     // freie Felder der naechsten Spalte
  60 MemWrite 8: 0x28a87e0             // ... schreiben
  62 MemRead 8: 0x0568125c00030751005  // naechster Durchlauf
```

Auflistung 5.5: Damenproblem – Mitrion-C Bandbreitengraph nach der Optimierung

einer separaten Bedingung untergebracht, um die Logiktiefe zu verringern. Die Berechnung der gleichen Test-Vorbelegungen benötigte nach der Optimierung 1 Stunde und 48 Minuten, wobei 50 Teilprobleme gleichzeitig verarbeitet werden können. Der zugehörige Bandbreitengraph und einige Debug-Ausgaben sind in Listing 5.5 dargestellt. Die kritische Schleife wird mit nun noch elf Takten Latenz angegeben. Werden zusätzlich die Debug-Ausgaben für den Schleifenrumpf betrachtet, kann der kritische Pfad mit zehn Takten Latenz abgelesen werden. Ein Takt wird zur Auswertung der Schleifenbedingung benötigt.

Es gibt drei wesentliche Unterschiede zur VHDL-Implementierung. Der VHDL-Entwurf ermöglicht die gleichzeitige Bearbeitung von über 140 Teilproblemen und arbeitet mit einer baumförmigen Verteilungsstruktur, um die Lösungsinstanzen mit Vorplazierungen zu versorgen. Die Mitrion-C-Lösung weist jeder Lösungsinstanz eine gewisse Anzahl von Vorbelegungen fest zu. Durch die unterschiedliche Verarbeitungszeit von Teilproblemen dominiert immer eine Lösungsinstanz (mit der längsten Rechenzeit) die Gesamtberechnung. Dadurch geht etwa die Hälfte der Verarbeitungsleistung verloren, wenn pro Durchlauf jeder Löser nur ein Teilproblem berechnet. Desweiteren sind die Lösungsinstanzen in der VHDL-Implementierung schneller.

Einen Optimierungsansatz bilden die Blockierungsvektoren und die noch freien Felder in einer Spalte, welche wie in Software-Algorithmen für jede Spalte gespeichert werden. Die derzeitige Mitrion-C-Implementierung nutzt an dieser Stelle Block-RAM. Das erhöht die Latenz der Schleife drastisch, da zum

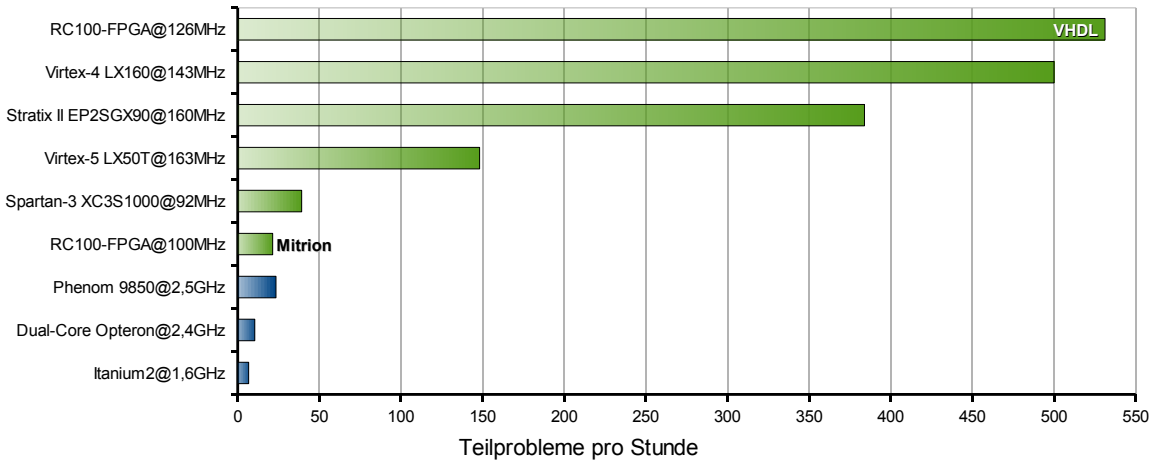


Abbildung 5.11: 26-Damenproblem – Leistungsvergleich (10 Stunden) von FPGAs und CPUs

einen die parallele Ausführung von Bedingungsprüfung und den beiden Zweigen nicht möglich ist und zum anderen die Zugriffe auf den Block-RAM bei Mitrion mehr Zeit als auf normale Register kosten. Die Block-RAM-Nutzung senkt allerdings auch den Bedarf an Flipflops, wodurch mehr Lösungsinstanzen implementiert werden können. Globale Blockierungsvektoren und eine andere Lösung zur Ermittlung der bereits gesetzten Damen wären Verbesserungsansätze. Allerdings müssten alle Block-RAM-Zugriffe in der WHILE-Schleife vermieden werden, um eine signifikante Leistungssteigerung zu erzielen. An dieser Stelle konnte der Mitrion-Support auch keine echten Optimierungsmöglichkeiten zur Erhöhung der Verarbeitungsleistung finden.

5.5.4 Performance

Da die Berechnungszeit eines Teilproblems stark von der zugehörigen Vorplatzierung abhängt, ist es schwierig, genaue Vergleiche zwischen Implementierungen zu ziehen. Der Damenproblem-VHDL-Core läuft auf den Virtex-4 FPGAs des RC100-Blades meist über einen längeren Zeitraum und ist an die Infrastruktur des Queens@TUD-Projektes angeschlossen. Ein dem Algorithmus von Jeff Somers ähnliches C-Programm wird genutzt, um zeitweise Prozessoren parallel dazu rechnen zu lassen. Damit kann die Performance von CPUs und der VHDL-Implementierung der RC100-FPGAs und anderer FPGAs gut verglichen werden. Ein gewisser Zufallsfaktor kommt aber dennoch hinzu, da jede Löserinstanz verschiedene Teilprobleme (mit unterschiedlichem Rechenaufwand) berechnet. Über einen ausreichend langen Zeitraum sollte dieser Faktor aber vernachlässigbar sein. Abbildung 5.11 zeigt die Messergebnisse für einen Zeitraum von zwölf Stunden. Test-Vorbelegungen wurden genutzt, um die Mitrion-C- und VHDL-Implementierung vergleichen zu können, wodurch letztendlich auch ein Gesamtvergleich aller Implementierungen möglich wird.

Das VHDL-Design zur Berechnung des 26-Damenproblems erreicht insbesondere auf den großen FPGAs deutliche Geschwindigkeitsgewinne im Vergleich zu auf CPUs laufenden Software-Lösungen. Die Implementierung mit Mitrion ist in etwa so schnell wie der AMD Phenom Vierkern-Prozessor. Wenn die Optimierungsvorschläge aus Abschnitt 5.5.3 (letzter Absatz) umgesetzt würden, wäre womöglich noch ein Faktor zwei bis drei an Geschwindigkeitsgewinn denkbar, dennoch wird das Mitrion-Design weit hinter einer VHDL-Implementierung des Damenproblems liegen.

6 Auswertung

In den vorangegangenen Kapiteln wurde die RASC-Plattform genauer untersucht und anhand von Fallbeispielen Praxistests durchgeführt. Auf diesen Fakten und Messergebnissen aufbauend soll überblicksweise das Leistungsvermögen des Systems eingeschätzt und auch Schwächen aufgedeckt werden. Zudem werden in Abschnitt 6.2 die mit den beiden Programmiermöglichkeiten Mitrion-C und VHDL gewonnenen Ergebnisse und Erfahrungen ausgewertet. Als Kriterien dienen die quantifizierbaren Metriken Entwicklungszeit und Performance und zusätzlich der Ease-of-Use, die Debugging-Möglichkeiten und die Fehleranfälligkeit. In Abschnitt 6.1 wird zuvor noch untersucht, welche Anwendungen zur Beschleunigung durch FPGA-basierte Systeme geeignet sind und welches Beschleunigungspotenzial theoretisch vorhanden ist. Als Ergebnis der Auswertung wird in Abschnitt 6.3 betrachtet, inwiefern sich der Implementierungsaufwand bei den erreichten Ergebnissen mit Mitrion-C und VHDL rechtfertigt und in welchem Verhältnis Entwicklungszeit und Performance stehen. Schließlich werden in Abschnitt 6.4 Mängel von SGI RASC angegeben und Verbesserungsvorschläge gemacht.

6.1 Anwendungsbeschleunigung durch FPGAs

Überlicherweise werden durch Zusatz-Hardware keine kompletten Anwendungen beschleunigt, sondern nur Routinen oder Algorithmen, welche die Gesamtlaufzeit dominieren. Um das Beschleunigungspotenzial und damit die erreichbare Performance ermitteln zu können, müssen vor der Umsetzung einige Faktoren untersucht werden:

- Vorhandensein von Schlüsselroutinen
- Intensität der Berechnung (Verhältnis von Berechnungen zu Kommunikation)
- Datenformat/Datenorganisation (Datentransfer zwischen Host und FPGA, Bereitstellungsaufwand)
- Spezialoperationen, z.B. Fließkommaoperationen oder spezielle mathematische Operationen

Ein nicht zu vernachlässigender Faktor ist der zusätzliche Kommunikationsaufwand zwischen dem Hostprogramm und dem Hardware-Beschleuniger. Dieser kann zwischen verschiedenen Anwendungen stark schwanken und in einigen Fällen sogar ein Flaschenhals sein, wodurch keine Beschleunigung der Ge-

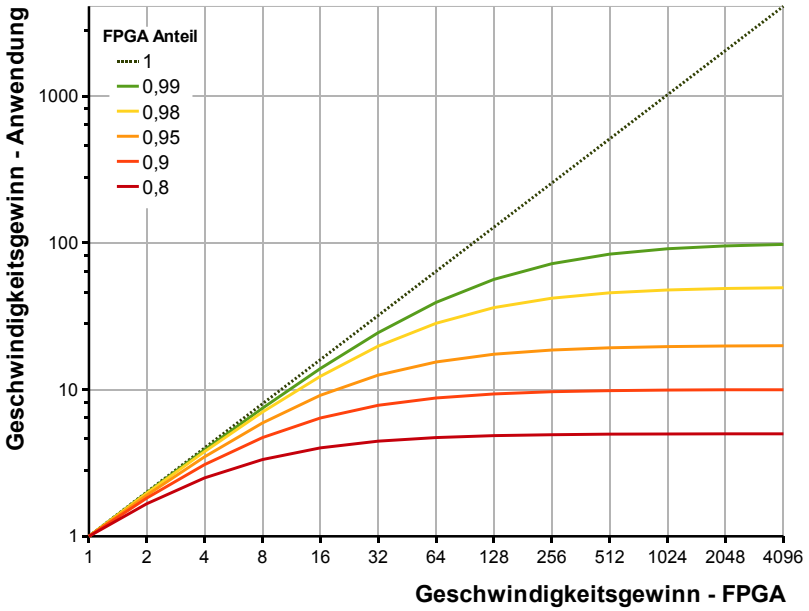


Abbildung 6.1: Geschwindigkeitsgewinn einer durch FPGAs beschleunigten Anwendung

samtanwendung möglich ist. Ein sehr hohes Beschleunigungspotenzial bieten Anwendungen mit vernachlässigbarem Kommunikationsaufwand und einer gut parallelisierbaren Schlüsselroutine, dessen Berechnungsaufwand die Gesamtlaufzeit dominiert.

Der Geschwindigkeitsgewinn einer Anwendung unterliegt auch bei der Verwendung von Hardware-Beschleunigern dem Gesetz von Amdahl. Das Diagramm 6.1 verdeutlicht die erreichbare Performance-Steigerung, wenn nur ein Teil (der Ausführungszeit) einer Anwendung durch den FPGA beschleunigt wird. Es ist zu erkennen, dass auch eine hohe Beschleunigung der Schlüsselroutine nur dann die Performance der Gesamtanwendung deutlich verbessern kann, wenn die Laufzeit des auf den FPGA ausgelagerten Teils zuvor die Gesamtrechenzeit dominiert hat.

6.2 Vergleich von Mitrion-C und VHDL (RASC-Programmierung)

Mit der Verwendung von Mitrion-C und VHDL werden in dieser Arbeit zwei vollkommen verschiedene Herangehensweisen zur Anwendungsbeschleunigung verglichen. In der Hochsprache Mitrion-C wird ein Algorithmus, wie in typischen prozeduralen Programmiersprachen, auf funktionaler Ebene beschrieben und durch einen Compiler (und dem Konzept des MVP, vgl. Abschnitt 4.1.2) in eine Hardwarebeschreibung überführt. Mit VHDL hingegen wird nicht der Algorithmus beschrieben, sondern die Hardware, welche den Algorithmus ausführen soll. Dementsprechend findet eine Algorithmusbeschreibung auf unterschiedlichem Abstraktionsniveau statt.

Die Verwendung von Mitrion-C zur Programmierung der RASC-Plattform bringt zwei grundlegende Einschränkungen mit sich. Dies ist zum einen die festgelegte Taktfrequenz des erzeugten Hardware-Designs auf 100 MHz und betrifft zum anderen die RASC-Kommunikationsmöglichkeiten, welche nicht in vollem Umfang genutzt werden können (siehe Abschnitte 5.1, 5.2.1 und 5.3).

Im Folgenden soll die Entwicklungszeit und die Performance der vorgenommenen Implementierungen der Fallbeispiele genauer betrachtet werden, um später Aufwand und Nutzen abschätzen zu können. Ease-of-Use, Debugging-Möglichkeiten und Fehleranfälligkeit sollen einen Einblick in den Programmiervorgang verschaffen, ohne einen zahlenmäßigen Vergleich der Sprachen vorzunehmen.

6.2.1 Entwicklungszeiten

Die Entwicklungszeiten können im Umfang dieser Arbeit nur abgeschätzt werden, da eine statistische Grundlage mit einer Vielzahl von Programmierern und Programmbeispielen nicht vorgenommen werden konnte. In die Betrachtung sollen zusätzlich noch die Einarbeitungszeiten mit einfließen. Neben den zeitlichen Unterschieden bei der Implementierung gibt es auch gemeinsame Entwicklungsaspekte, welche am Ende des Abschnitts kurz besprochen werden.

Die für die Micro-Benchmarks angefallenen Entwicklungszeiten können verhältnismäßig genau angegeben werden, da sie im Umfang dieser Arbeit erstellt wurden und keinen komplexen Algorithmus implementieren. Für beide Sprachen bestand die Aufgabe im Wesentlichen in der Ansteuerung der hardwareseitigen Schnittstelle der RASC-Plattform (Core Services). Bei Mitrion-C fällt für diesen Punkt nahezu keine Entwicklungszeit an, da die für den Nutzer sichtbare Schnittstelle die Übergabeparameter der Main-Funktion sind. Diese werden wie normale Typen (hier Speicher, Streams oder Variablen) gehandhabt.

Bei der Verwendung von VHDL hingegen müssen zum einen Wrapper-Module und Konfigurationsdateien erzeugt und zum anderen die Ansteuerung der Schnittstelle zu den Core Services (vgl. Abschnitte 5.2.1 und 5.1.1) vorgenommen werden. Der zeitliche Aufwand für die reine Implementierung ist mit etwa einer Stunde Arbeitszeit zu bewerten, wenn das Algorithmus-Konfigurations-Tool (vgl. Abschnitt 3.4.4) verwendet wird und die Schnittstellenspezifikation bekannt ist. Andernfalls muss zuvor die Spezifikation studiert und Tests durchgeführt werden, um eine funktionsfähige Hardware-Umsetzung zu erhalten. Dies hat bei jeder Kommunikationsvariante einige Tage in Anspruch genommen, da insbesondere das Timing der von den Core Services bereitgestellten Signalen problematisch war. Die finalen Implementierungen können jedoch mit wenigen Modifikationen auf andere Beispiele übertragen werden, wodurch ein Großteil des Entwicklungsaufwandes für diese Schnittstellenbeschreibung nur einmal anfällt. Die Datenmanipulation lässt sich dann in VHDL ebenso schnell und knapp wie in Mitrion-C beschreiben.

Nach Angaben der Entwickler (vgl. [PNS09]) konnte eine erste funktionsfähige VHDL-Damenproblem-Implementierung in nur drei Stunden erstellt werden. Das aktuelle Design ist aber in einem iterativen Prozess entstanden, bei der jede Überarbeitung erneute Vorüberlegungen und Neuimplementierungen mit sich gebracht hat. Hinzu kommt die Anbindung an die RASC-Plattform, welche ungefähr einen Tag in Anspruch genommen hat und durch Änderungen am Design und neue Anforderungen mehrfach angepasst wurde. Aus einem C-Programm abgeleitet konnte eine erste funktional korrekte Implementierung in Mitrion-C in etwa zwei Tagen erstellt werden. Anschließend wurden noch mehrere Tage in die Optimierung investiert. Weitere angedachte Verbesserungen sollten in wenigen Tagen umgesetzt werden können.

Die Mitrion-C-MD5-Implementierung konnte nach Angaben der Entwickler (vgl. [IJ08]) in etwa drei Arbeitstagen erstellt werden. Anschließend wurden nur wenige Optimierungen durchgeführt. Eine erste funktionierende VHDL-Implementierung benötigte ca. drei Wochen und wurde seither in mehreren Iterationen verbessert, wobei noch immer Optimierungen ausstehen.

Die Einarbeitungszeit in Mitrion-C ist (insbesondere bei Vorkenntnissen in der Programmiersprache C) deutlich geringer als in VHDL, da funktional und von der Hardware abstrahiert programmiert wird. VHDL-Programmierung benötigt hingegen fundiertes Wissen über die Hardware. Um auf ein vergleichbares Niveau zu kommen, wird die Einarbeitungszeit für VHDL mit einem halben Jahr und für Mitrion-C mit einem Monat abgeschätzt. Unter diesen Vorraussetzungen wird ein VHDL-Entwurf zur Umsetzung eines Algorithmus etwa viermal soviel Zeit beanspruchen, wie ein vergleichbares Mitrion-C-Programm, wenn sich der Algorithmus mit beiden Sprachen umsetzen lässt. Die Grundgedanken zum Entwurf eines Algorithmus, der sich gut zur parallelen Abarbeitung eignet, sind für beide Programmiermöglichkeiten gleichermaßen zeitaufwendig und können mit steigender Komplexität des Problems die Implementierungszeit übertreffen.

Eine deutliche Verringerung der Entwicklungszeit kann (insbesondere bei Hardwarebeschreibung) durch die Wiederverwendung von Entwürfen (engl. Design Reuse) erwirkt werden. VHDL-Implementierungen bieten zudem durch die Abbildung des Algorithmus auf Hardwarestrukturen mehr Optimierungsmöglichkeiten, wodurch sich auch die Entwicklungszeit deutlich verlängern kann. In Mitrion-C entstehen Optimierungen häufig durch Ausprobieren und anschließender Bewertung, ob eine Verbesserung erreicht wurde. Dies kommt durch das Compiler-generierte Design, welches sich durch geringe Modifikationen des Mitrion-C-Codes stark verändern kann. Gute Implementierungen und Optimierungen basieren in beiden Sprachen jedoch meistens auf Erfahrungswerten.

Für beide Programmiermöglichkeiten kommt zur Algorithmusimplementierung noch der Zeitaufwand für die Hardware-Synthese hinzu, welche für die RASC-Plattform etwa eine Stunde für kleine und bis

zu mehreren Tagen für große Designs in Anspruch nimmt. An dieser Stelle muss noch erwähnt werden, dass Timing-Anforderungen innerhalb der RASC Core Services sehr lange Synthesezeiten hervorrufen können, während der eigentliche Algorithmus bereits erfolgreich verdrahtet wurde. Auch die vielen Komponenten, aus denen der MVP aufgebaut ist erhöhen die Synthesezeit zu vergleichbaren VHDL-Designs. Für Software ist der Übersetzungsaufwand hingegen deutlich geringer und liegt für in dieser Arbeit betrachtete Algorithmen im Millisekundenbereich.

Schließlich muss der durch den FPGA beschleunigte Algorithmus noch in die Hostanwendung integriert werden. Hierzu müssen zum einen die zu übertragenden Daten bereitgestellt (ggf. umformatiert) und zum anderen die Ansteuerung der RASC-FPGAs vorgenommen werden. Funktion der *rasclib*-Bibliothek (vgl. Tabelle 3.2) ermöglichen die Kommunikation, wobei Mitrion zusätzlich ein eigenes API anbietet. Je nach Bereitstellungsaufwand der Daten liegt der Zeitaufwand im Bereich einiger Stunden. Von SGI bereitgestellte Beispiele oder bereits geschriebene Programme helfen dabei den Aufwand zu verringern, da die Ansteuerung verschiedener RASC-Kommunikationsvarianten sehr ähnlich ist.

6.2.2 Ease-of-Use

Bedienkomfort oder Bedienbarkeit sind im Grunde durch den Programmierer subjektiv gefühlte Eigenschaften. Um diese dennoch bewerten zu können, werden sie anhand einiger Fakten und Beispielen veranschaulicht. Zudem sollte beachtet werden, dass die Erwartungen gegenüber einer Hardwarebeschreibungssprache sich von denen einer Hochsprache unterscheiden.

Um schnell eine einfache Anwendung auf die RASC-Plattform zu portieren, eignet sich Mitrion sehr gut, weil zum einen die Einarbeitungszeit in die vom Syntax C-ähnliche Sprache recht kurz ist und zum anderen die Schnittstelle zum vorliegenden System einfach gehalten wurde. Mit VHDL muss hingegen erst noch die Schnittstelle beschrieben und an das Verilog-Wrapper-Modul angebunden werden, was insbesondere bei der Implementierung erster Beispielandwendungen Mühe bereitet und das ausführliche Studieren des RASC-Benutzerhandbuchs ([SGI08]) erfordert.

Von einer sich an C anlehenden Hochsprache wird neben der Ähnlichkeit zum Syntax auch ein gewisses Repertoire an Standardkonstrukten erwartet. Mitrion-C besitzt hier einige Eigenheiten, wie z.B.:

- Anweisungsblöcke (Schleifen und Bedingungen) haben Rückgabewerte
- einmalige Variablenzuweisungen in einem Block (engl. single assignment)
- Unterscheidung zwischen schleifenabhängigen und -unabhängigen Variablen
- Listen und Vektoren anstelle von Feldern (engl. arrays)

Rückgabewerte von Anweisungsblöcken werden hinter diesen in Klammern angegeben und ermögli-

chen, neben schleifenabhängigen Variablen, das Propagieren von Werten aus einem Block. Anlehnend an die C-Syntax wären return-Anweisungen u.U. intuitiver. Es gibt die Schleifentypen FOR, FOREACH und WHILE. Die FOR-Schleife muss mindestens eine schleifenabhängige Variable enthalten, während die FOREACH-Schleife keine enhalten darf. Anweisungen wie *break* oder *continue* stehen nicht zur Verfügung. Desweiteren können keine einfachen Bedingungen formuliert werden, ohne den Alternativ-Zweig anzugeben. Während einige dieser Eigenschaften von Mitrion-C der Fehlervermeidung dienen, können sie auch als Einschränkungen betrachtet werden, welche eigentlich durch den Compiler oder eine Synthese gehandhabt werden sollten.

VHDL hingegen erfüllt die Erwartungen an eine Hardwarebeschreibungssprache in vollem Umfang. Sogar Hochsprachenelemente, wie z.B. Schleifen, Funktionen oder abstrakte Datentypen, können verwendet werden. Die Beschränkung auf einmalige Signalzuweisung gilt nur außerhalb von Prozessen, innerhalb wird der letzte zugewiesene Wert übernommen. Bei einem bedingten Anweisungsblock werden im nicht angegebenen alternativen Zweig keine Signale verändert. Es gibt in VHDL keine Rückgabewerte von Anweisungsblöcken, da Signale in einer Komponente global sind und Variablen innerhalb eines Prozesses gelten.

Das Damenproblem ist ein Beispiel, welches in Mitrion-C nicht so beschrieben werden kann, dass es gute Performance aufweist. Gewisse, teilweise bewusste Einschränkungen im Sprachumfang begründen diesen Umstand. Beispielsweise gibt es im Vergleich zu VHDL oder typischen Programmiersprachen keine zu einem Feld (engl. array) vergleichbare Struktur. Mitrion-Listen können nur sequentiell abgearbeitet werden, während Mitrion-Vektoren kein indiziertes Schreiben von einzelnen Elementen erlauben. Eine Umgehungslösung (engl. workaround) dafür bietet die Nutzung von Block-RAM, welcher allerdings die parallele Abarbeitung in Mitrion-C-Programmen deutlich einschränkt. Allgemein sind Datenstrukturen in VHDL flexibler und können auch vom Programmierer definert werden. Eine Möglichkeit in Mitrion-C Strukturen zu beschreiben sind Tuple. Auf Elemente in Tuples kann jedoch nicht indiziert oder durch die Angabe einer Variablenbezeichung zugegriffen werden. Sie dienen lediglich dem Zusammenfassen mehrerer Variablen, um gerade bei Rückgabewerten von Blöcken das Programm übersichtlicher zu gestalten.

Die wesentliche Vereinfachung der Mitrion-C-Programmierung im Vergleich zur Hardware-Beschreibung betrifft die Bereitstellungen von Daten und die automatische Verwaltung von Datenabhängigkeiten. Dadurch kann typisch sequentiell programmiert werden, während dem Nutzer die eigentliche datengetriebene Abarbeitungsreihenfolge von Operationen verborgen bleibt. Bei VHDL muss der Programmierer für die zeitlich korrekte (taktgenaue) Bereitstellung der Daten sorgen. Das folgt allerdings daraus, dass es sich um Hardware- und keine Algorithmus-Beschreibung handelt. Wird beispielsweise die Verwendung von Block-RAM betrachtet, kann diese mit Mitrion-C durch einfache Funktionsaufrufe bewerkstelligt

werden. In VHDL muss ein Block oder eine Komponente beschrieben werden, welche über Adress-, Enable- und Daten-Signale angesteuert wird. Bei lesendem Zugriff muss einen Takt bevor der Wert bereit stehen soll, die entsprechende Adresse angelegt werden, während bei schreibendem Zugriff die Adresse im gleichen Takt wie das Datum angelegt wird. Bei Mitrion-C braucht der Programmierer sich um solche Hardware-Detailes nicht zu kümmern.

Die Hardware-Synthese wird in beiden Fällen von der Kommandozeile gestartet. Bei Mitrion kann mit dem Aufruf einer Java-Anwendung und entsprechender Parameterübergabe sowohl simuliert als auch compiliert oder synthetisiert werden. Optional können Parameter wie der Syntheseaufwand, Ausgabe- und temporäres Datenverzeichnis auch in einer Benutzeroberfläche eingestellt werden. Der Nutzer hat auf die Paramter der Hardware-Synthese keinen direkten Einfluss, kann für den gesamten Prozess aber zwischen drei Aufwandsstufen wählen. Dies ist vorallem ohne Hintergrundwissen über die zugrundeliegende Hardware einfach zu entscheiden, aber für erfahrene Benutzer ärgerlich; gerade wenn ein Design nur knapp nicht geroutet werden konnte. Eine genauere Anpassung ist nur auf einem Umweg zu erreichen, wobei die Mitrion-Anwendung zu Beginn der Synthese abgebrochen wird und die temporären Dateien im RASC-Arbeitsablauf weiterverarbeitet werden (z.B. mit einem Makefile). Dieser wird bei der Verwendung von VHDL genutzt und ermöglicht eine genaue Anpassung der Hardware-Synthese. Ein Makefile und eine weitere Datei, in welcher die High-Level-Synthese-Optionen aufgelistet sind, dienen dazu. Zudem wird neben ISE mit Synplify Pro ein zweites Synthese-Tool unterstützt – Mitrion nutzt ausschließlich ISE. Während der RASC-Arbeitsablauf zu keinem Zeitpunkt mehr als acht GByte benötigt, überschreitet Mitrion diese Grenze bei großen Designs und kann dementsprechend nicht mehr auf kleinen Systemen mit weniger Hautspeicher in akzeptabler Zeit ausgeführt werden.

Entwicklungsumgebungen (z.B. Xilinx ISE) und Editoren mit Syntax-Hervorhebung erleichtern den Entwurfsprozess mit VHDL. Die in früheren Versionen von Mitrion enthaltene Entwicklungsumgebung gehört in der aktuellen Version nicht mehr zum Paket und es muss in einem beliebigen Editor programmiert und auf Kommandozeile übersetzt werden.

Mitrion-C ermöglicht dem Programmierer einen schnellen Einstieg zur Nutzung von rekonfigurierbarer Hardware, ohne Wissen über diese zu benötigen. Der Mangel an Beschreibungsformen (Konstrukten) lässt allerdings nicht in jedem Fall die effiziente Programmierung des Algorithmus zu. Mit VHDL wird hingegen die Hardware zur Abarbeitung eines Algorithmus beschrieben, was in vielen Fällen zu einer effizienteren Umsetzung führt. Mitrion-C kann mit einigen guten Konstrukten wie die Vektor- oder Listenerzeugung über Bereiche oder der FOREACH-Schleife zur einfachen Parallelisierung überzeugen, enttäuscht aber gleichermaßen durch einen Mangel an Beschreibungsformen (z.B. keine Felder, continue-, break- und exit-Anweisung).

6.2.3 Fehleranfälligkeit und Debugging-Möglichkeiten

Beim Programmieren kann zwischen verschiedenen Arten von Fehlern unterschieden werden. Lexikalische und syntaktische Fehler werden bereits zur Übersetzungszeit vom Compiler erkannt, während semantische und logische Fehler meist erst zur Laufzeit festgestellt werden. Durch einen fehlerhaften Algorithmus bedingte Fehler können in beiden Sprachen gleichermaßen auftreten.

Die Anzahl lexikalischer und syntaktischer Fehler steigt überlicherweise konstant mit der Länge des Quellcodes. Durch den längeren Quellcode bei der Umsetzung vergleichbarer Algorithmen werden solche Fehler beim VHDL-Entwurf häufiger vorkommen als bei Mitrion-C-Programmen. Allerdings gibt es im Verleich zu Mitrion-C für VHDL Entwicklungsumgebungen und Editoren mit VHDL-Syntax-Hervorhebung, was diese Art von Fehler wiederum verringert. Einige semantische Fehler, wie z.B. nicht erlaubte Zuweisungen von Signal- bzw. Variablentypen, werden ebenso bereits während der Übersetzung erkannt und können meist schnell behoben werden. Bei Mitrion-C wird Variablen ohne explizite Typangabe automatisch anhand der ersten Zuweisung ein Typ zugeordnet. Dies verringert einerseits Fehler während der Übersetzung, kann aber zu inhaltlichen Fehlern führen. Ein Beispiel hierfür ist die Addition von Zahlen, dessen Summe bei Mitrion-C ein Bit breiter als der größere der beiden Summanden ist. VHDL ist hingegen eine streng typenorientierte Sprache und ein verwendetes, aber nicht deklariertes Signal führt während der Syntaxprüfung zu einem Fehler.

Eine fehlerhafte VHDL-Implementierung folgt häufiger daraus, dass Daten nicht im vorgesehenen Takt am gewünschten Signal anliegen oder es wurde vergessen einem Signal einen Initialwert zu übergeben. Solche Fehler im Hardware-Design können mit Mitrion-C nicht auftreten, da Datenabhängigkeiten automatisch gehandhabt werden und einer Variablen bei der Deklaration ein Wert übergeben werden muss. Es können noch viele andere Fehler vorkommen, welche allerdings in ihrer Gesamtheit hier nicht aufgezählt werden sollen. Festzuhalten ist jedoch, dass mit VHDL mehr Fehler dieser Art entstehen können, da auf einer niedrigeren Abstraktionsebene als mit Mitrion-C programmiert wird und die Beschreibung des Kontrollflusses aufwändiger ist. Zudem werden bei Mitrion-C z.B. durch Unterscheidung von FOREACH- und FOR-Schleife, sowie das Single Assignment-Prinzip und komplett ausformulierte Bedingungszweige Fehler frühzeitig (zur Übersetzungszeit) erkannt.

Eine inkorrekte Implementierung und damit ein semantischer oder logischer Fehler kann mit beiden Sprachen anhand einer Simulation bereits aufgedeckt werden. Die Mitrion-Simulation (vgl. Abschnitt 4.1.3) ist mit der Verwendung von Watch-Anweisungen einer Hochsprachenfehlersuche sehr ähnlich. Beim Aufruf einer solchen Anweisung wird der Inhalt der beobachteten Variable auf der Kommandozeile ausgegeben. Zudem kann mit dem grafischen Simulator die Struktur des Programmes angesehen werden, wodurch das Aufdecken funktionaler Fehler auch optisch möglich ist. Die Fehlersuche in einem

VHDL-Entwurf ist aufwändiger als bei Mitrion-C-Programmen und wird üblicherweise vor der Synthese mit einem HDL-Simulator (vgl. Abschnitt 4.2.2) vorgenommen. Dabei können die an den Signalen und Ports der untersuchten Komponenten und Unterkomponenten anliegenden Werte und entsprechende Signalverläufe angesehen werden, um Fehler im Entwurf aufzudecken.

Während eine fehlerfreie Simulation mit Mitrion während der Tests stets zu einer korrekten Hardware-Umsetzung führte, war dies bei den VHDL-Entwürfen nicht immer der Fall. Zur Simulation mit VHDL wird üblicherweise eine Testbench erstellt, welche das Verhalten der Schnittstelle beschreibt. Da für das Verhalten der Core Services an den Schnittstellen zum Algorithmus von SGI keine Testbench zur Verfügung gestellt wird, muss diese laut der Spezfikation in [SGI08] vom RASC-Nutzer programmiert werden. Eine fehlerhafte oder unvollständige Beschreibung der Testumgebung kann jedoch auch zu einer inkorrekten oder nicht funktionierenden Hardware-Umsetzung führen.

Eine weitere Möglichkeit der Fehlersuche ist mit den Debug-Registern gegeben. In beiden Sprachen können Signale bzw. Variablen an diese gebunden und auf Hostseite, während der Laufzeit deren Inhalt abgefragt werden. Dies kann auch taktweise mit dem GNU-Debugger (siehe Abschnitt 3.4.3) vorgenommen werden, wobei dieser das Direct Streaming nicht unterstützt.

6.2.4 Performance

Die Performance der auf die RASC-Plattform portierten Fallbeispiele in Mitrion-C- und VHDL wurde im einzelnen bereits in Kapitel 5 ausgewertet. Tabelle 6.1 zeigt die besten Messergebnisse der Beispielanwendungen und die Geschwindigkeitsgewinne (S_p) zwischen den Implementierungen im Überblick. Die Performance-Auswertung betracht einen RC100-FPGA (XC4VLX200), wobei der Referenz-CPU für MD5-Brute-Force der Core2Duo-Prozessor und für das Damenproblem der Phenom-Prozessor ist.

	CPU	Mitrion	VHDL	$\mathrm{Sp} = \frac{\mathrm{Mitrion}}{\mathrm{CPU}}$	$\mathrm{Sp} = \frac{\mathrm{VHDL}}{\mathrm{CPU}}$	$\mathrm{Sp} = \frac{\mathrm{VHDL}}{\mathrm{Mitrion}}$
MD5-Brute-Force (M Hashes/s)	88	288	1173	3,3	13,3	4,1
26-Damenproblem (Teilprobl./Std.)	23	21	531	0,9	23,1	25,3

Tabelle 6.1: Performance-Überblick

Mitrion-C-Umsetzungen benötigen im Durchschnitt laut Angaben des Herstellers Mitrionics in etwa doppelt soviel FPGA-Fläche (Overhead durch die Processing Elements des MVP) wie vergleichbare VHDL-Implementierungen, was bei einem auf Parallelität basierendem Design die erreichbare Performance auf die Hälfte verringert. Diese optimistische Schätzung konnte durch die Beispiel-Implementierungen des MD5-Brute-Force-Algorithmus und des 26-Damenproblems nicht bestätigt werden. Für die VHDL-Implementierungen wurden in allen Fallbeispielen bessere Messwerte als mit vergleichbaren Mitrion-Umsetzungen erreicht.

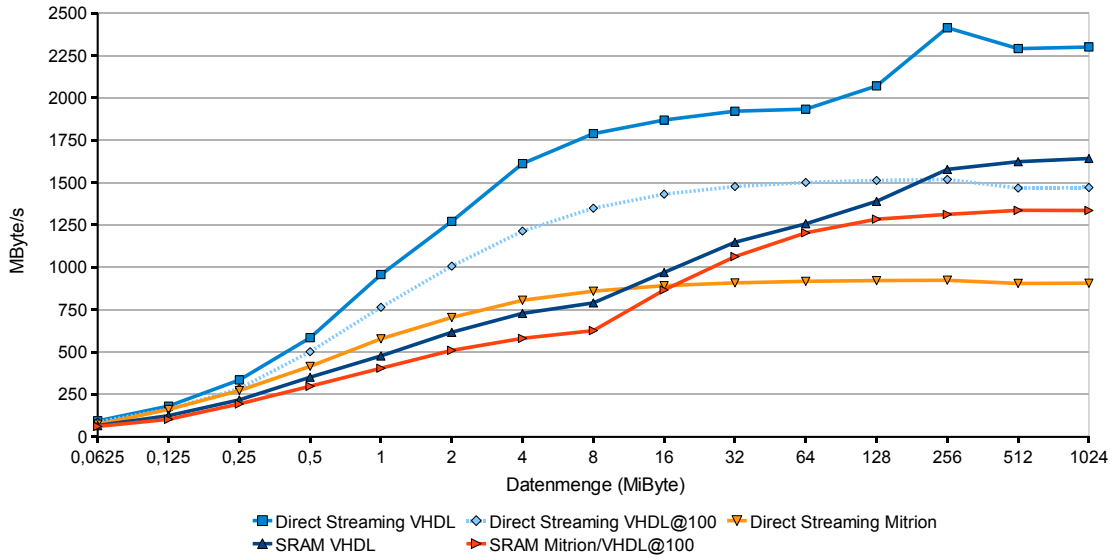


Abbildung 6.2: Zusammenfassung des Datendurchsatzes mit Direct I/O

Ist die Anwendung gut durch Pipeline- und FOREACH-Strukturen auf der innersten Schleifenebene in Mitrion-C zu beschreiben (MD5-Brute-Force), kann ein Geschwindigkeitsgewinn gegenüber einer auf einem CPU laufenden, hoch optimierten Software von bis 3,3 gemessen werden. Die VHDL-Umsetzung ist jedoch noch einmal um einen Faktor 4,1 schneller als Mitrion. Lässt sich die Anwendung nicht gut in Mitrion-C beschreiben (Damenproblem), so ist mitunter kein Geschwindigkeitsgewinn gegenüber CPUs zu verzeichnen, wohingegen die VHDL-Umsetzung mit einem Beschleunigungsfaktor von rund 23 gemessen werden konnte. Wie beim Hardware-Entwurf muss auch bei der Mitrion-C-Programmierung auf möglichst flache Strukturen (geringe Verschachtelung von Bedingungen und Schleifen) geachtet werden, um eine gute Performance zu erreichen.

Auch die Ergebnisse der Durchsatzmessungen liegen mit Mitrion hinter den vergleichbaren VHDL-Implementierungen, was zumindest bei der Verwendung von SRAM auf die geringere Taktung des MVP zurückzuführen ist. Der maximal erreichbare Direct-Streaming-Durchsatz der mit 100 MHz getakteten VHDL-Implementierung ist dennoch deutlich schneller als die Mitrion-Variante, weshalb hier von einem Fehler im MVP-Design ausgegangen werden kann. Abbildung 6.2 zeigt noch einmal die für Direct I/O gemessenen Werte für die SRAM-Kommunikation und das Direct Streaming.

Der SRAM-Kontroller der Core Services scheint derzeit noch ein Flaschenhals zu sein, zumal nach den theoretischen Angaben 3,2 GByte/s Datendurchsatz möglich sind und mit Direct Streaming ein Datendurchsatz von 2,4 GByte/s praktisch erreicht wurde. Die Durchsatzbeschränkung tritt bei der Kommunikation zwischen Host und SRAM auf. Bei einem Test wurden zwischen Algorithmus-Block und SRAM die maximal erreichbaren 3,2 GByte/s Datendurchsatz auch praktisch gemessen (vgl. 5.1.2). Zudem wurde die SRAM-Kommunikation ohne die Verarbeitung der Daten durch den Algorithmus gemessen und

erreichte nur einen maximalen Durchsatz von 1,65 GByte/s. Anhand dieser Ergebnisse und dem Aufbau des Datenpfades (vgl. Abbildung 3.4) kann sich der Flaschenhals nur in einem Teil des Speicherkontrollers der Core Services befinden.

6.3 Kosten-Nutzen-Abschätzung und Fazit

Als Ergebnis der Auswertung werden in diesem Abschnitt Kosten und Nutzen der Anwendungsportierung abgeschätzt und damit die beiden quantifizierbaren Metriken Entwicklungszeit und Performance in einen Zusammenhang gestellt. Eine Wirtschaftlichkeitsuntersuchung oder pauschale Abschätzung der Kosten und Nutzen ist aufgrund mangelnder praxisrelevanter Beispiele im Umfang dieser Arbeit nicht möglich. Allein die Damenproblem-Implementierung arbeitet über einen längeren Zeitraum am Queens@TUD-Projekt ([PNS]) mit, jedoch nicht in einem kommerziellen Rahmen.

Als Grundlage der Abschätzung dienen die Fallbeispiele MD5-Brute-Force und 26-Damenproblem. Die Kosten werden zeitlich in Form der Entwicklungszeit t_E angenommen, während der Nutzen auf den Geschwindigkeitsgewinn S_p (vgl. Tabelle 6.1) abgebildet werden kann. Wird die Ersetzung des Referenzsystems durch das Beschleunigersystem betrachtet, kann zudem die zeitliche Amortisierung t_A berechnet werden. Während der Anwendungsportierung finden dann keine Berechnungen statt. Abhängig vom Geschwindigkeitsgewinn der beschleunigten Anwendung, kann der Ausfall der Berechnungen nach einer Zeit t_A kompensiert werden. Tabelle 6.2 zeigt die verwendeten Metriken und die zugehörige Amortisationszeit (vgl. Formel 6.1) für jeweils zwei Entwicklungsstufen der Beispielanwendungen.

$$t_A = t_E \cdot \sum_{n=1}^{\infty} \frac{1}{{S_p}^n} \tag{6.1}$$

Die in der Tabelle angegebenen Entwicklungszeiten (t_E) sind Schätzwerte (vgl. Abschnitt 6.2.1) und es wird nur mit einem FPGA des RC100-Blades als Beschleuniger gerechnet, weil nur eine CPU als

	VHDL			Mitrion		
	t_E	S_p	t_A	t_E	S_p	t_A
Damenproblem(1)	7 Tage	11,4	16,2 Stunden	2 Tage	0,5	∞
Damenproblem(2)	14 Tage	22,7	15,5 Stunden	5 Tage	0,9	∞
MD5-Brute-Force(1)	10 Tage	6,6	42,9 Stunden	3 Tage	3,3	31,3 Stunden
MD5-Brute-Force(2)	18 Tage	13,3	35,1 Stunden	3 Tage	3,3	31,3 Stunden
Durchschnitt ∅	12 Tage	13,5	23 Stunden	3 Tage	2,0	72 Stunden

t_E ... Entwicklungszeit; t_A ... Amortisationszeit;
S_p ... Geschwindigkeitsgewinn gegenüber CPU

Tabelle 6.2: Entwicklungszeit und Performance

Referenz verwendet wurde. Die Nutzung beider FPGAs verdoppelt den angegebenen Geschwindigkeitsgewinn S_p und verringert die Amortisationszeiten t_A. In dieser Arbeit konnten nur Beispiele mit recht kurzer Entwicklungszeit betrachtet werden, wodurch schon nach geringer Laufzeit der beschleunigten Implementierung eine zeitliche Amortisierung auftritt. Der Geschwindigkeitsgewinn ist außerdem ein Maß dafür, wieviele Referenzsysteme betrieben werden müssten, um die gleiche Performance zu erreichen.

Um eine Wirtschaftlichkeitsuntersuchung durchzuführen, müssten noch die Gesamtkosten (Entwicklungsingenieur, Beschaffungskosten des neuen Systems, Energiekosten, etc.) betrachtet und mit dem durchschnittlichen Gewinn in ein Verhältnis gesetzt werden. Aus diesen Ergebnissen und der durchschnittlichen Lebenserwartung eines HPC-Systems von fünf Jahren kann dann abgeschätzt werden, ob sich der Kauf des Beschleunigersystems rentiert. Zudem müssten schlecht quantifizierbare Faktoren, wie z.B. Ease-of-Use, Fehleranfälligkeit oder Qualität der Implementierung der Vollständigkeit halber in die Untersuchung mit einfließen.

Zu beachten ist auch, dass mehrere Systeme höhere laufende Kosten verursachen. Werden beispielsweise folgende Daten angenommen, können Energiekosten und CO_2-Ausstoß über einen beliebigen Zeitraum berechnet werden:

- 15 Eurocent pro Kilowattstunde (15ct/kWh)
- 400 Gramm CO_2-Ausstoß pro Kilowattstunde (400g/kWh)
- FPGAs mit 25 Watt Verlustleistung
- Referenz-CPUs mit 65 Watt Verlustleistung

Im Zeitraum von 200 Tagen (4800 Stunden) verbraucht damit ein FPGA 120kW. Dies entspricht Energiekosten von 18 € und einem CO_2-Ausstoß von 48 kg. Wird ein Beschleunigungsfaktor 20 für die FPGA-Implementierung angenommen, müssten anstelle dessen 20 Referenz-CPUs arbeiten, um die gleiche Rechenleistung zu erbringen. Dabei würden Energiekosten von 936 € anfallen und 2496 kg CO_2 ausgestoßen.

Im positiven Fall für Mitrion kann der zu beschleunigende Algorithmus in Mitrion-C effizient beschrieben werden, benötigt dann im Vergleich zur VHDL-Implementierung ein Sechstel der Entwicklungszeit und bringt ein Viertel der Performance. In ungünstigen Fällen kann mit Mitrion jedoch kein Geschwindikeitsgewinn erreicht werden. Ein Grund für die vergleichsweise geringen Geschwindigkeitsgewinne ist die langsamere Taktung des Mitrion-Designs. Über die Taktverwaltung der Virtex-4 FPGAs (DCMs) und FIFOs mit unterschiedlichem Lese- und Schreibtakt wäre auch eine feinere Einstellung der Taktung des Mitrion-Algorithmus denkbar und könnte zumindest als experimentelle Funktion angeboten werden.

Weitere Gründe ergeben sich aus dem Mangel an Beschreibungsmöglichkeiten (vgl. Abschnitte 6.2.2 und 5.5.3).

Mitrion ermöglicht mit ca. drei Wochen Einarbeitungszeit und einer Woche Entwicklungszeit einen schnellen Einstieg und erreichte in den Messungen im besten Fall einen Geschwindigkeitsgewinn von 3,3 gegenüber der schnellsten gemessenen CPU. Es ist jedoch nicht jeder Algorithmus effizient in Mitrion-C umsetzbar. Im allgemeinen ist das Ziel einen Algorithmus unter Verwendung einer FOREACH-Schleife mit einer Pipeline-Struktur auf der innersten Schleifenebene zu beschreiben. Während dies bei MD5-Brute-Force (vgl. 5.4.2) möglich ist und zu einem Geschwindigkeitsgewinn führt, lässt sich das Damenproblem einfach, allerdings nicht effizient beschreiben und führt zu einer vergleichsweise schlechten Performance. In einigen Fällen muss ein anderer Ansatz gefunden werden, um eine Beschleunigung mit Mitrion-C zu erreichen, wodurch sich allerdings auch der Entwicklungsaufwand deutlich erhöht.

Kurze Entwicklungszeiten zur schnellen Portierung eines Algorithmus in VHDL erreichen nicht das Leistungsvermögen ausgereifter Implementierungen, sind mitunter dennoch schneller als eine vergleichbare Mitrion-C-Implementierung. Die prinzipielle Herangehensweise zur Algorithmusportierung ist mit beiden Programmiermöglichkeiten ähnlich, da ein guter paralleler Algorithmus auch die Grundlage für Geschwindigkeitsgewinne bildet. Letztendlich ist Mitrion-C ein interessanter Programmieransatz, der nicht ausgereift ist, aber in zukünftigen Versionen noch Leistungssteigerungen erwarten lässt. Mit einer guten VHDL-Implementierung kann bei geeigneter Algorithmuswahl (parallel und gut auf FPGAs umsetzbar) immer mit Geschwindigkeitsgewinnen gerechnet werden.

6.4 Verbesserungsansätze der RASC-Plattform

Während der Implementierung der Fallbeispiele und den entsprechenden Performance-Messungen konnten einige Mängel an SGI RASC festgestellt werden. Zudem werden hier noch einige Verbesserungsvorschläge angegeben, welche den Ease-of-Use der gesamten Programmierplattform erhöhen würden.

Die „Supplemental Clock“ (ein zusätzlich durch die Core Services zur Verfügung gestelltes einstellbares Taktsignal) kann aus Gründen des Timings nur bedingt verwendet werden. Eine typische Variante Daten zwischen Taktdomänen zu übertragen sind FIFOs mit asynchronem Schreib- und Lesetakt. Allerdings konnte mit solchen durch den Xilinx Fifo Generator v4.4 erstellten FIFOs das Timing nicht erreicht werden. Alternativ wurde im Algorithmus selber eine DCM initialisiert und damit erfolgreich eine Taktanpassung vorgenommen.

Die Geräteverwaltung für die FPGAs (devmgr) hat mitunter mehrere Wochen den Dienst untersagt. Die falsche Angabe eines Dateinamens beim Hinzufügen oder Verändern eines Algorithmus führte z.B. dazu,

dass die gesamte Registrierung und Abfrage der Algorithmen nicht mehr funktionierte (Fehler wurde bereits behoben). Zusätzlich traten andere bisher ungeklärte Fehler auf, die mitunter den Absturz des kompletten Systems zur Folge hatten.

Um den mit einer HDL arbeitenden RASC-Nutzer zu unterstützen, wäre eine Testbench, welche das Verhalten der Core Services an der Schnittstelle zum Algorithmus beschreibt, sehr hilfreich und würde Fehler und entsprechend unnötige Synthesevorgänge vermeiden. Die Testbench könnte neben den Konfigurationsdateien und Verilog-Wrapper-Modulen durch das Algorithmus-Konfigurations-Tool generisch erzeugt werden.

Weitere Verbesserungsansätze betreffen die in Abschnitt 3.3 vorgestellten Kommunikationsvarianten, deren Funktionalität erweitert werden könnte:

Algorithm Defined Registers: Der Host wird derzeit nicht über eine Änderung der Register durch den Algorithmus informiert. Eine entsprechende funktionale Erweiterung der rasclib-Bibliothek würde das Polling über ein zusätzliches Debug-Register (vgl. 5.3.1) überflüssig machen.

SRAM-Kommunikation: Mit der aktuellen Version der RASC Core Services ist es nicht möglich unabhängig vom FPGA-Algorithmus auf den SRAM zuzugreifen. Es können ausschließlich einmal, während der Ausführung des Algorithmus, Daten vom Host gesendet und empfangen werden. Eine unabhängige Host-SRAM-Kommunikation würde einen dauerhaften Betrieb des FPGA-Algorithmus ermöglichen. Dementsprechend müssten auch zusätzliche Kontrollsignale in die Core Services integriert werden, um den Algorithmus über die Schreib- und Lesevorgänge des Host zu informieren.

Direct Streaming Wie bei der SRAM-Kommunikation ist es nur möglich einmal pro Algorithmus-Durchlauf Daten zu senden oder zu empfangen. Sinnvoll wäre es, wenn ein Streaming-Port mehrfach Datenströme empfangen oder senden könnte, ohne das zwischendurch der Algorithmus beendet werden muss.

7 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde die SGI RASC Plattform untersucht und mit Mitrion-C und VHDL zwei Programmiermöglichkeiten verglichen. Hierfür wurde in einem analytischen Teil die Beschleuniger-Technologie vorgestellt und dabei insbesondere auf die verwendeten Virtex-4 FPGAs eingegangen. Zudem wurden Aufbau und Kenndaten des RC100-Blades vorgestellt, sowie die Software- und Hardware-Programmierschnittstelle untersucht. Das RC100-Blade integriert sich sehr gut in die Architektur der Altix 4700 und ist mit der vollen Bandbreite von 6,4 GByte/s an das NUMAlink4-Netzwerk angeschlossen.

Da die Fließkommaverarbeitungsleistung ein typisches Vergleichsmaß im Hochleistungsrechnen ist, wurde diese in einer theoretischen Betrachtung für mehrere Virtex-FPGAs abgeschätzt. Die von SGI RASC unabhängige Untersuchung kann auf verschiedene FPGA-basierte Beschleuniger-Systeme angewendet werden und zeigt, dass FPGAs durchaus mit aktuellen Prozessoren mithalten können, andere Hardware-Beschleuniger, wie z.B. GPUs in diesem Bereich jedoch bessere Performance bieten.

Mit dem Damenproblem und dem MD5-Brute-Force-Algorithmus konnten zwei komplexere Beispiele auf die RASC-Plattform portiert und deutliche Leistungssteigerungen gegenüber aktuellen Prozessoren verzeichnet werden. Die Kommunikation mit dem RC100-Blade wurde anhand von Mikro-Benchmarks getestet und erreichte unter der Verwendung von Direct I/O und Direct Streaming einen maximalen Datendurchsatz von etwa 2,4 GByte/s. Die Messungen des Datendurchsatzes zwischen Host und SRAM erreichten mit 1,65 GByte/s gerade die Hälfte des theoretisch Möglichen. Der FPGA kann hingegen mit voller theoretischer Bandbreite mit dem SRAM kommunizieren, was zu der Schlussfolgerung führt, dass der Flaschenhals im Speicherkontroller der Core Services liegt. Die Latenz von indizierten Speicherzugriffen des FPGA auf den lokalen SRAM wurde mit 80 ns gemessen. Zusätzlich konnten die Memory Mapped Registers (auf den Speicher des Hosts abgebildete FPGA-Register) neben der Parameterübergabe und Debugging-Zwecken auch als Variante der Datenübertragung verwendet werden. Dadurch wurde die Anbindung der Damenproblem-Implementierung an die Infrastruktur von Queens@TUD erst möglich und es konnte ein Beitrag zum Fortschritt des Projektes geleistet werden.

Alle Fallbeispiele wurden sowohl mit Mitrion-C als auch mit VHDL auf die RASC-Plattform portiert. Basierend auf diesen Implementierungen wurden die verschiedenen Metriken Entwicklungszeit, Ease-

of-Use, Fehleranfälligkeit, Debugging-Möglichkeiten und Performance bewertet. Mitrion-C offenbarte dabei insbesondere Schwächen im Sprachumfang, was bei betroffenen Algorithmen zu einer vergleichsweise schlechten Performance führt. Der maximale Geschwindigkeitsgewinn eines RC100-FPGAs gegenüber einer Referenz-CPU ist laut Messungen 3,3 für Mitrion-C und 23,1 für VHDL. Allgemein ist festzustellen, dass die Mitrion-C-Implementierungen in allen Messungen schlechtere Performance bieten, als die vergleichbaren VHDL-Implementierungen. Zur Leistungsbewertung der Hardware-Implementierungen wurden in der Programmiersprache C Benchmarks geschrieben. Die Messungen wurden entweder über einen längeren Zeitraum oder aber unter der Verwendung von Bash-Skripten mehrfach durchgeführt, um zeitweise Schwankungen zu interpolieren.

Eine Kosten-Nutzen-Abschätzung bringt schließlich die quantifizierbaren Metriken Entwicklungsaufwand und Performance in einen Zusammenhang und verdeutlicht, wann sich eine Anwendungsportierung auf die RASC-Plattform lohnt. Als Fazit bleibt, dass sich der VHDL-Entwurf einer geeigneten Anwendung immer lohnt, während mit Mitrion-C mitunter keine Geschwindigkeitsgewinne gegenüber aktuellen CPUs erreicht werden können, wodurch sich auch der im Durchschnitt geringere Entwicklungsaufwand nicht immer rechtfertigen lässt. Die vergleichsweise geringe Leistungsaufnahme von FPGAs kann allerdings schon bei geringen Geschwindigkeitsgewinnen Energiekosten verringern.

Neben dem noch vorhandenem Optimierungspotenzial der vorgenommenen Implementierungen bietet auch die RASC-Plattform selber noch Verbesserungsmöglichkeiten, was insbesondere die Kommunikationsvarianten betrifft. Einige Ansätze wurden bereits in der Auswertung erläutert.

An dieser Stelle bieten sich nun Vergleiche zu anderen FPGA-basierten Programmierplattformen an. Sog. In-Socket-Accelerators ermöglichen z.B. eine engere Kopplung der Beschleuniger-Chips und CPUs, was geringere Latenzen und höhere Bandbreiten ermöglichen soll. Zudem wurde mit Mitrion-C nur eine Hochsprache untersucht, welche FPGA-Systeme als Zielplattform hat. Handel-C und Impulse-C sind weitere Programmiersprachen, welche die Nutzung von FPGA-basierten Systemen verfolgen. Diese arbeiten mit einer Teilmenge der C-Funktionalität und C-Erweiterungen zum parallelen Entwurf. Eine weitere Herausforderung betrifft die Portierung einer echten Anwendung (nicht aus der internen Forschung), welche bereits Rechenzeit auf den am ZIH installierten Hochleistungsrechnern beansprucht. Schließlich sollte noch untersucht werden, inwiefern dynamische partielle Rekonfiguration die Flexibilität von FPGAs erhöht und Nutzbarkeit im Bereich des Hochleistungsrechnen verbessert.

Für den zukünftigen Erfolg FPGA-basierter Systeme im Hochleistungsrechnen könnte die OpenFPGA-Initiative beitragen. Diese wird von einer Vielzahl von Firmen und Organisationen unterstützt und arbeitet an einem einheitlichen Industriestandard zur Integration von rekonfigurierbarer Technologie in leistungsstarke Rechensysteme. Eine allgemeine API-Spezifikation soll, anbieterunabhängig und portierbar

zwischen FPGA-Systemen, die Kommunikation mit Nutzeranwendungen oder Bibliotheken ermöglichen. Derzeit muss in Frage gestellt werden, ob mit der Open Computing Language (OpenCL), welche als gemeinsame Programmiersprache für Grafikbeschleuniger und DSPs entwickelt wird, auch FPGAs programmiert werden können. Der hier vorgestellte Hochsprachenansatz bietet zudem auch nicht die Geschwindigkeitsgewinne, die von einem Hardware-Beschleuniger erwartet werden.

Letztendlich bleibt abzuwarten, inwiefern sich FPGAs im HPC etablieren können. Hierfür ist auch eine Sensibilisierung der Nutzer notwendig, da die Portierung von Anwendungen immer einen Entwicklungsaufwand mit sich zieht.

Abbildungsverzeichnis

2.1 Typische Architektur eines FPGA 9
2.2 FPGA Logikblock und Schaltmatrix (vgl. [Wik09]) 10
2.3 Paritätsbit für 32-Bit-Wert: 4-Eingangs-LUT (links), 6-Eingangs-LUT (rechts) 10
2.4 Konfigurierbarer Logikblock (CLB) der Virtex-4 FPGAs (vgl. [Xil08c]) 14
2.5 Stark vereinfachter Virtex-4-Slice (ohne Carry- und Speicher-Logik, vgl. [Xil08c]) . . . 14
2.6 Energieeffizienz von FPGAs und GPUs 20

3.1 Altix 4700 Rechenknoten 22
3.2 RC100-Blade (vgl. [SGI08]) 24
3.3 RASC FPGA-Modul (vgl. [SGI08]) 24
3.4 RASC Core Services (vgl. [SGI08]) 25
3.5 SGI RASC – Abstraktionsebenen (vgl. [SGI08]) 29

4.1 Entwicklungszyklus – Mitrion SDK auf SGI RASC (vgl. [Mit08b]) 36
4.2 Integration von Mitrion in SGI RASC (vgl. [Mit08a]) 38
4.3 Mitrion-C Simulation mit GUI 41
4.4 Vergleich von Hardware- und Softwareentwicklung 44
4.5 Hardware-Synthese im Überblick ([Hoc08], Kapitel 1) 49

5.1 Direct IO Streaming Durchsatz aller Knoten (256 MiByte) 54
5.2 Durchsatz über den SRAM des RC100-Blades 57
5.3 Direct Streaming Durchsatz 60
5.4 Memory Mapped Registers – Kommunikationsschema 62
5.5 Memory Mapped Registers – Durchsatzmessung 63
5.6 MD5 Brute Force – VHDL-Entwurf 66
5.7 MD5 Brute Force – Performance Vergleich 71
5.8 Eine Lösung des Acht-Damenproblems 72
5.9 Damenproblem Backtracking-Algorithmus 73
5.10 Infrastruktur und RASC-Anbindung von Queens@TUD 75
5.11 26-Damenproblem – Leistungsvergleich (10 Stunden) von FPGAs und CPUs 79

6.1 Geschwindigkeitsgewinn einer durch FPGAs beschleunigten Anwendung 82
6.2 Zusammenfassung des Datendurchsatzes mit Direct I/O 90

Tabellenverzeichnis

2.1 Kenndaten einiger Virtex-FPGAs 17
2.2 FP-Performance von Virtex-5-FPGAs im Vergleich zu Opteron Prozessor 18
2.3 Virtex-FPGAs: FP-Performance Multiplikation-Addition (64 Bit / 32 Bit) 18

3.1 Intel Itanium 2 Montecito[Int06] - Cache-Hierarchie 22
3.2 Ausgewählte Funktionen der *rasclib*-Bibliothek 30

5.1 MD5 Brute-Force: FPGA Ressourcen-Nutzung – VHDL 68
5.2 MD5 Brute-Force: FPGA Ressourcen-Nutzung – Mitrion 69
5.3 Leistungsvergleich der MD5-Hardware-Implementierungen 70
5.4 Maximale Suchdauer MD5-basierter Passwörter (VHDL-Implementierung) 71
5.5 Lösungen einiger N-Damenprobleme 74
5.6 26-Damenproblem: Ressourcen-Nutzung – VHDL 76

6.1 Performance-Überblick 89
6.2 Entwicklungszeit und Performance 91

Auflistungsverzeichnis

4.1 Mitrion-C Beispielprogramm . . . 37
4.2 Mitrion Bandbreitengraph . . . 42
5.1 Mitrion-C-Programm zum Test der SRAM-Anbindung . . . 56
5.2 Mitrion-C-Programm zum Test des Direct Streaming . . . 59
5.3 Umsetzung der MD5-Operationen (Runden 1 und 2) in VHDL . . . 67
5.4 Mitrion-C MD5-Pipeline . . . 69
5.5 Damenproblem – Mitrion-C Bandbreitengraph nach der Optimierung . . . 78
A.1 Steuerung der SRAM-Kommunikation mit VHDL . . . 107
A.2 SRAM Latenzmessung mit VHDL . . . 109
A.3 Ausschnitte des C Programms zur Durchsatzmessung des SRAM . . . 110
A.4 Direct Streaming VHDL-Implementierung . . . 112
A.5 Messschleife – Direct Streaming . . . 112
A.6 Memory Mapped Registers: VHDL-Implementierung . . . 114
A.7 Messschleife – Memory Mapped Registers . . . 116
A.8 VHDL MD5-Pipeline (Ausschnitt) . . . 118
A.9 MD5 Hash-Vergleich in VHDL . . . 119
A.10 MD5 Klartextrückgewinnung in VHDL . . . 119
A.11 Datenübertragung MD5-Brute-Force Hostprogramm . . . 121
A.12 Erster Versuch einer Mitrion-C-Implementierung des Damenproblems . . . 123
A.13 Bandbreitengraph Damenproblem Mitrion-C (1) . . . 125
A.14 Optimierte Mitrion-C-Implementierung des Damenproblems . . . 126
A.15 Damenproblem: Native C-Funktionen . . . 128
A.16 Damenproblem: Java Interface . . . 130

Literaturverzeichnis

[AD09] ARBEITER, Stefan ; DEEG, Matthias: Bunte Rechenknechte - Grafikkarten beschleunigen Passwort-Cracker. In: *c't 2009, Heft 6* (2009)

[Dev07] DEVINE, Christophe: *MD5 Source Code*. Webseite. Oktober 2007. – `http://xyssl.org/code/source/md5`

[Hoc08] HOCHBERGER, Christian: *Hardware-Synthese für eingebettete Systeme*. Vorlesungsfolien. 2008. – `http://www.mr.inf.tu-dresden.de`

[IJ08] ILSCHE, Thomas ; JUCKELAND, Guido: First experiences with SGI RASC at TU Dresden / Center for Information Services and High Performance Computing, TU Dresden, Germany. 2008. – Forschungsbericht

[Int06] INTEL: *Dual-Core Update to the Intel Itanium 2 Processor Referenz Manual For Softare Development and Optimization*. Revision 0.9, Januar 2006

[KC07] KREINDLER, Danny ; CORPORATION, Altera. *How to implement double-precision floating-point on FPGAs*. White Paper. Oktober 2007

[Mä08] MÄDER, Andreas: *VHDL Kompakt*. Universität Hamburg, MIN-Fakultät, Department Informatik, Oktober 2008

[Mit08a] MITRIONICS: *Low Power Hybrid Computing for Efficient Software Acceleration*. White Paper. 2008. – `http://www.mitrion.com/?document=Hybrid-Computing-Whitepaper.pdf`

[Mit08b] MITRIONICS: *Mitrion User's Guide*, 2008. – `http://forum.mitrionics.com/uploads/Mitrion_Users_Guide.pdf`

[PNS] PREUSSER, Thomas B. ; NÄGEL, Bernd ; SPALLEK, Rainer G.: *Queens@TUD Projekt*. – `http://queens.inf.tu-dresden.de/`

[PNS09] PREUSSER, Thomas B. ; NÄGEL, Bernd ; SPALLEK, Rainer G.: Putting Queens in Carry Chains / Institut für Technische Informatik, TU Dresden, Germany. 2009. – Forschungsbericht. `ftp://ftp.inf.tu-dresden.de/pub/berichte/tud09-03.pdf`. – ISSN 1430–211X

[Pre05] PREUSSER, Thomas B. *VHDL - Ein Überblick*. Vorlesungsfolien. Juni 2005

[Riv92] RIVEST, Ronald L.: *The MD5 Message-Digest Algorithm*. Webseite. April 1992. – `http://tools.ietf.org/html/rfc1321`

[SGI08] SGI: *Reconfigurable Application-Specific Computing User's Guide*, 2008. – `http://techpubs.sgi.com/library/manuals/4000/007-4718-007/pdf/007-4718-007.pdf`

[Smi96] SMITH, Douglas J.: VHDL & Verilog Compared & Contrasted - Plus Modeled Example Written in VHDL, Verilog and C / VeriBest Incorporated. 1996. – Forschungsbericht

[Som02] SOMERS, Jeff: *The N Queens Problem - a study in optimization*. 2002. – http://www.jsomers.com/nqueen_demo/nqueens.html

[SSWW08] STRENSKI, Dave ; SIMKINS, Jim ; WALKE, Richard ; WITTIG, Ralph. *Revaluating FPGAs for 64-bit Floating-Point Calculations*. White Paper. Mai 2008

[Str07] STRENSKI, Dave. *FPGA Floating Point Performance - a pencil and paper evaluation*. White Paper. Januar 2007

[Wag08] WAGNER, Michael: *Grafikprozessoren als Hardwarebeschleuniger - Vergleich der Ansätze von NVIDIA und AMD zur Nutzung der Ressourcen aktueller Grafikprozessoren für allgeimeine Anwendungen*. Zentrum für Informationsdienste und Hochleistungsrechnen, Diplomarbeit, Oktober 2008

[Wan98] WANNEMACHER, Markus: *Das FPGA-Kochbuch*. MITP-Verlag, 1998. – ISBN 3-8266-2712-1

[Wik09] WIKIPEDIA: *Field Programmable Gate Array*. April 2009. – http://de.wikipedia.org/wiki/Fpga

[WY05] WANG, Xiaoyun ; YU, Hongbo: How to Break MD5 and Other Hash Functions / Shandong University, Jinan 250100, China. 2005. – Forschungsbericht

[Xil] XILINX: *Getting started with FPGAs*. Webseite. – http://www.xilinx.com/company/gettingstarted/index.htm

[Xil07] XILINX: *Virtex-4 Family Overview, DS112 (v3.0)*. Produkt Spezifikation. September 2007. – http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[Xil08a] XILINX: *Floating-Point Operator v4.0, DS335*. Produkt Spezifikation. April 2008. – http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

[Xil08b] XILINX: *Virtex-4 FPGA Configuration User Guide*. v2.10, April 2008. – http://www.xilinx.com/support/documentation/user_guides/ug071.pdf

[Xil08c] XILINX: *Virtex-4 FPGA User Guide*, 2008. – http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[Xil09a] XILINX: *Virtex-5 Family Overview, DS100 (v5.0)*. Produkt Spezifikation. Februar 2009. – http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[Xil09b] XILINX: *Virtex-6 Family Overview, DS150 (v1.1)*. Produkt Spezifikation. Mai 2009. – http://www.xilinx.com/publications/prod_mktg/Virtex6_Overview.pdf

A Quellcodes

A.1 SRAM-Schnittstelle

Auflistung A.1: Steuerung der SRAM-Kommunikation mit VHDL

```
-- extractor VERSION: 1.4
-- extractor CS: 2.2
-- extractor SRAM:sram0_in 524288 128 sram[0] 0x0000 in  u stream
-- extractor SRAM:sram1_out 524288 128 sram[1] 0x0000 out u stream
-- extractor REG_IN:op_count 19 u alg_def_reg[0][18:0]
-- extractor REG_IN:inc_val 8 u alg_def_reg[0][47:32]

-- Schnittstelle ...
-- Signaldefinitionen ...

-- parameters set with algorithm defined register
op_count  <= adr0(18 downto 0);
inc_value <= adr0(47 downto 32);

-- request values from SRAM0
mem0_rd_cmd_vld <= sram0_rd_cmd_vld;
mem0_rd_addr <= sram0_alg_offset(9) & mem0_idx & "0000";
read_mem: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge
    sram0_rd_cmd_vld <= '0';
    if (rst = '1') then --synchronous reset
      mem0_idx <= (others => '0');
      read_done <= '0';
    else
      if (read_done = '0') AND (sram0_rd_busy = '0') then
        sram0_rd_cmd_vld <= '1';
      end if;

      if sram0_rd_cmd_vld = '1' then
        mem0_idx <= mem0_idx + 1;
      end if;

      if mem0_idx = op_count then
        read_done <= '1';
      end if;
    end if;
  end if;
end process;

-- write values from sram0 to sram1
mem1_wr_be   <= X"ffff";
mem1_wr_addr <= sram1_alg_offset(9) & mem1_idx & "0000";
mem1_wr_cmd_vld <= sram1_wr_cmd_vld;
write_mem: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge
```

```
    sram1_wr_cmd_vld <= '0';
    if (rst = '1') then --synchronous reset
      mem1_wr_data <= (others => '0');
      mem1_idx <= (others => '0');
      write_done <= '0';
    else
      if (sram0_rd_data_vld = '1') AND (write_done = '0') then
        -- split input into two 64 Bit values and add inc_value to both
        mem1_wr_data(127 downto 80) <= sram0_rd_data(127 downto 80);
        mem1_wr_data(79 downto 64) <= sram0_rd_data(79 downto 64) + inc_value;
        mem1_wr_data(63 downto 16) <= sram0_rd_data(63 downto 16);
        mem1_wr_data(15 downto 0) <= sram0_rd_data(15 downto 0) + inc_value;
        sram1_wr_cmd_vld <= '1';
      end if;

      if sram1_wr_cmd_vld = '1' then
        mem1_idx <= mem1_idx + 1;
      end if;

      if mem1_idx = op_count then
        write_done <= '1';
      end if;
    end if;
    alg_done <= write_done;
  end if;
end process;
```

Auflistung A.2: SRAM Latenzmessung mit VHDL

```
proc_latency:process(clk)
begin
if (clk'event and clk = '1') then -- rising edge

  mem2_wr_cmd_vld <= '0';
  mem2_rd_cmd_vld <= '0';
  mem2_wr_addr <= (others => '0');
  mem2_rd_addr <= (others => '0');

  if (rst = '1') then --synchronous reset
    mem2_wr_data(63 downto 0) <= (others => '0');
    ctr <= (others => '0');
    mem2_wr_done <= '0';
    mem2_rd_start <= '0';
    mem2_rd_done <= '0';
    -- extractor REG_OUT:mem2_latency 64 u debug_port[4]
    debug4 <= (others => '0');
  else
    ctr <= ctr + 1;

    if mem2_wr_done = '0' AND sram2_wr_busy = '0' then
      mem2_wr_data(11 downto 0) <= ctr; -- write current ctr
      mem2_wr_data(63 downto 12) <= (others => '0');
      mem2_wr_cmd_vld <= '1';
      mem2_wr_done <= '1';
    end if;

    if mem2_rd_start = '0' AND mem2_wr_done = '1' AND sram0_rd_busy = '0' then
      mem2_rd_cmd_vld <= '1';
      mem2_rd_start <= '1';
      debug4(23 downto 12) <= ctr;
    end if;

    if sram2_rd_data_vld = '1' AND mem2_rd_done = '0' then
      debug4(35 downto 24) <= ctr;
      debug4(11 downto 0) <= sram2_rd_data(11 downto 0);
      mem2_rd_done <= '1';
    end if;

  end if;
end if;
end process;
```

Auflistung A.3: Ausschnitte des C Programms zur Durchsatzmessung des SRAM

```
unsigned long *in;
unsigned long *out;

int main(int argc, char *argv[]){
  // evaluate command line arguments
  ...
  // variables declaration
  ...

  // allocate memory
  if(buffers_make(&arguments) < 0){
    printf("buffers_make failed\n");
    return -1;
  }

  // initialize input values (arguments.size*2) 64 Bit values
  for (i = 0; i < arguments.size*2; i++) {
    *(in+i)=0;
  }

  // special argument for VHDL implementation
  if(arguments.vhdl == TRUE){
    if(arguments.nop == TRUE){
      op_count = 0;
    }else{
      op_count = arguments.size - 1;
    }
    res = rasclib_algorithm_reg_write(al_desc, "op_count", &op_count, 1,
        RASCLIB_IMMED_CMD);
    if (res != RASCLIB_SUCCESS) {
      fprintf(stderr,"reg write of op_count failed at %d: %d\n", __LINE__, res);
      rasclib_perror("reg write of op_count", res);
      return 1;
    }
  }

  // send value to be added by algorithm to every 64 Bit value
  res = rasclib_algorithm_reg_write(al_desc, "inc_val", &inc, 1,
      RASCLIB_IMMED_CMD);
  if(res != RASCLIB_SUCCESS){
    fprintf(stderr,"reg write of alg_inc_val failed at %d: %d\n", __LINE__, res);
    rasclib_perror("reg write of alg_inc_val", res);
    return 1;
  }

  start = gtod(); // get start time
  // loop for measuring throughput
  for(i = 0; i<arguments.cycles;i++){
    res = rasclib_algorithm_send(al_desc, "sram0_in", in, arguments.num_bytes);
    if(res != RASCLIB_SUCCESS) {
      fprintf(stderr,"send failed at %d: %d\n", __LINE__, res);
      return 1;
    }

    rasclib_algorithm_go(al_desc); // start algorithm

    res = rasclib_algorithm_receive(al_desc, "sram1_out",out,arguments.num_bytes)
        ;
    if(res != RASCLIB_SUCCESS){
      fprintf(stderr,"recv failed at %d: %d\n", __LINE__, res);
      return 1;
    }
```

```
    rasclib_algorithm_commit(al_desc, NULL);
    rasclib_algorithm_wait(al_desc);
  }
  end = gtod(); // get end time

  if(arguments.vhdl == TRUE){
    rasclib_algorithm_reg_read(al_desc, "mem2_latency", &debug1,1,
        RASCLIB_IMMED_CMD);
    rasclib_algorithm_commit(al_desc, NULL);

    printf("mem2_latency vector = 0x%lx - ",debug1);
    printf("busy error vector = 0x%lx\n",debug2);
    latency=(unsigned int)(((debug1&0xFFF000000)>>24)-((debug1&0xFFF000)>>12));
    printf("SRAM access latency (1) = %d clock cycles (%d ns @200MHz) \n",latency
        ,latency*5);
  }

  buffers_free(arguments.io);
  rasclib_huge_clear();
  return 0;
}

// allokate memory depending on IO method
int buffers_make(struct cargs *args){
  if (RASCLIB_DIRECT_IO == (*args).io) {
    in = (unsigned long *) rasclib_huge_alloc((*args).num_bytes);
    out = (unsigned long *) rasclib_huge_alloc((*args).num_bytes);

    if(in == NULL || out == NULL) {
      buffers_free((*args).io);
      rasclib_huge_clear();
      return -1;
    }
    }else{
      in = (unsigned long *) malloc((*args).num_bytes);
      out = (unsigned long *) malloc((*args).num_bytes);

      if (in == NULL || out == NULL) {
        buffers_free((*args).io);
        return -1;
      }
    }
  return 0;
}

void buffers_free(int iomethod){
  if(RASCLIB_DIRECT_IO == iomethod){
    rasclib_huge_free(in);
    rasclib_huge_free(out);
  }else{
    free(in);
    free(out);
  }
}
```

A.2 Direct Streaming

Auflistung A.4: Direct Streaming VHDL-Implementierung

```
-- extractor CS: 2.2
-- extractor VERSION:1.3
-- extractor STREAM_IN:strm_in 0 0
-- extractor STREAM_OUT:strm_out 0 0
-- extractor REG_IN:inc_val 16 u alg_def_reg[0][15:0]

-- Schnittstelle ...
-- Signaldefinitionen ...

-- parameter set with algorithm defined register
inc_value <= adr0(15 downto 0);

-- start reading, when stream data are ready and algorithm is ready
strm_in_0_rd_en <= strm_in_data_rdy AND rd_en;
strm: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge
    rd_en <= '0';

    strm_out_0_data(127 downto 80) <= strm_in_data(127 downto 80);
    strm_out_0_data(79 downto 64) <= strm_in_data(79 downto 64) + inc_value;
    strm_out_0_data(63 downto 16) <= strm_in_data(63 downto 16);
    strm_out_0_data(15 downto 0) <= strm_in_data(15 downto 0) + inc_value;

    if (rst = '1') then --synchronous reset
      strm_out_0_data_vld <= '0';
      strm_out_0_data_last <= '0';
      strm_out_0_flush <= '0';
      strm_out_0_data <= (others => '0');
      alg_done <= '0';
    else
      if strm_out_almost_busy = '0' AND strm_in_complete='0' then
        rd_en <= '1';
      end if;
      strm_out_0_data_vld <= strm_in_vld;

      -- end of stream, send last value
      if strm_in_complete = '1' then
        strm_out_0_data_last <= '1';
        strm_out_0_flush <= '1';
      end if;

      alg_done <= strm_out_flushed;
    end if;
  end if;
end process;
```

Auflistung A.5: Messschleife – Direct Streaming

```
  starttime = gtod();
  for(i=0;i<arguments.cycles;i++){
    cop_desc = rasclib_cop_open(arguments.algID, arguments.io);
    if (cop_desc == RASCLIB_FAIL) {
      fprintf(stderr,"open failed at %d: %d\n", __LINE__, cop_desc);
      rasclib_perror("open", res);
      return 1;
    }

    res = rasclib_cop_reg_write(cop_desc,"inc_val",&inc,1,RASCLIB_IMMED_CMD);
```

```
    if (res != RASCLIB_SUCCESS) {
      fprintf(stderr,"reg write of inc_val failed at %d: %d\n", __LINE__, res);
      rasclib_perror("reg write of inc_val", res);
      return 1;
    }

    starttime2 = gtod();
    // send input stream data
    res = rasclib_cop_send(cop_desc, "strm_in", in, arguments.num_bytes);
    if (res != RASCLIB_SUCCESS) {
      fprintf(stderr,"send failed at %d: %d\n", __LINE__, res);
      rasclib_perror("send", res);
      return 1;
    }

    //  Signal input stream complete
    rasclib_cop_send_complete(cop_desc, "strm_in");
    if (res != RASCLIB_SUCCESS) {
      fprintf(stderr,"send_complete failed at %d: %d\n", __LINE__, res);
      rasclib_perror("send", res);
      return 1;
    }

    rasclib_cop_go(cop_desc); // start fpga algorithm

    // receive output stream data
    res = rasclib_cop_receive(cop_desc, "strm_out", out, arguments.num_bytes);
    if (res != RASCLIB_SUCCESS) {
      fprintf(stderr,"recv failed at %d: %d\n", __LINE__, res);
      rasclib_perror("receive", res);
      return 1;
    }

    rasclib_cop_commit(cop_desc, NULL);
    rasclib_cop_wait(cop_desc);
    endtime = gtod();
    rasclib_cop_close(cop_desc);
  }

  rasclib_resource_return(arguments.algID, 1);
  rasclib_resource_release(1, resrv_name);

  if(arguments.cycles > 1){
    time = endtime - starttime;
  }else{
    time = endtime - starttime2;
  }
  // print results to stdout
  printf("%f MByte/s (%f MiByte) throughput ",
                (float)(arguments.cycles*(arguments.num_bytes/time/1000000.0)),
                (float)(arguments.cycles*(arguments.num_bytes/time/1024/1024)));
```

A.3 Memory Mapped Registers

Auflistung A.6: Memory Mapped Registers: VHDL-Implementierung

```
-- Register 32 is used for flags (read and write by host and FPGA possible)
-- extractor REG_IN:finish_alg 1 u alg_def_reg[32][63]
-- extractor REG_IN:adr_used 5 u alg_def_reg[32][4:0]

-- Registers 0 to 15 are used as Register-Input (16 64Bit-values at once)
-- extractor REG_IN:fat_in 1024 u alg_def_reg[0][1023:0]

-- Registers 16 to 31 are used as Register-Output (16 64Bit-values at once)
-- extractor REG_IN:fat_out 1024 u alg_def_reg[16][1023:0]

-- Schnittstelle ...
-- Signaldefinitionen ...

-- manage input data
proc_input: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge
    fifo_put <= '0';
    FOR i IN 0 TO 15 LOOP
      if adr_updated(i) = '1' then
        fifo_din <= adr(i);
        fifo_put <= '1';
      end if;
    END LOOP;
  end if;
end process;

-- get register update latency
proc_debug: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge
    if (rst = '1') then --synchronous reset
      debug(1) <= (others => '0');
      debug(2) <= (others => '0');
      debug(3) <= (others => '0');
      debug(4) <= (others => '0');
      ctr_delay <= (others => '0');
    else
      ctr_delay <= ctr_delay + 1;
      if adr_updated /= 0 then
        debug(1)(59 downto 0) <= debug(1)(47 downto 0) & ctr_delay;
        debug(2)(59 downto 0) <= debug(2)(47 downto 0) & debug(1)(59 downto 48);
        debug(3)(59 downto 0) <= debug(3)(47 downto 0) & debug(2)(59 downto 48);
        debug(4)(59 downto 0) <= debug(4)(47 downto 0) & debug(3)(59 downto 48);
      end if;
    end if;
  end if;
end process;

-- manage rasc output
proc_rasc: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge
    if (rst = '1') then --synchronous reset
      rasc_out_full <= '0';
      ctr_rout <= (others => '0');
      ctr_polled <= (others => '0');

      debug(5) <= (others => '0');
```

```
      debug(6) <= (others => '0');
      debug(7) <= (others => '0');
    else
      if rasc_out_full = '1' then
        -- register read done by host
        if adr_polled /= 0 then
          ctr_polled <= ctr_polled + 1;
          -- get polling latency
          debug(5)(59 downto 0) <= debug(5)(47 downto 0) & ctr_delay;
          debug(6)(59 downto 0) <= debug(6)(47 downto 0) & debug(5)(59 downto 48)
              ;
          debug(7)(59 downto 0) <= debug(7)(47 downto 0) & debug(6)(59 downto 48)
              ;
        end if;
      end if;

      -- user defined number of ADRs polled
      if ctr_polled = adr_used then
        rasc_out_full <= '0';
        ctr_polled <= (others => '0');
      end if;

      if fifo_vld = '1' then
        if rasc_out_full = '0' then
          ctr_rout <= ctr_rout + 1;
        end if;
        if ctr_rout = adr_used-1 then
          rasc_out_full <= '1';
          ctr_rout <= (others => '0');
        end if;
      end if;
    end if;
  end if;
end process;
fifo_rd_en <= '1' when rasc_out_full = '0' AND fifo_empty = '0' else '0';

-- manage output data
proc_output: process(clk)
begin
  if (clk'event and clk = '1') then -- rising edge

    FOR i IN 0 to 15 LOOP
      adr_wr_data(i) <= fifo_dout;
    END LOOP;

    adr_wr <= (others => '0');  -- do not write to output registers
    if rasc_out_full = '0' AND fifo_vld = '1' then
      adr_wr(conv_integer(ctr_rout(3 downto 0))) <= '1';
    end if;

  end if;
end process;
-- finish algorithm execution intiated by host
alg_done <= '1' when finish_alg = '1' else '0';
```

Auflistung A.7: Messschleife – Memory Mapped Registers

```
0  rasclib_cop_go(cop_desc); // Start Bitstream Engine

   start = gtod();
   for(ctr=0;ctr<arguments.cycles;ctr++){
     unsigned long *fatin;
5    unsigned long *fatout;
     fatin=in;
     fatout=out;
     for(i=0;i<arguments.fifo/arguments.adr_used;i++){
       // Send RASC FPGA Input
10     res = rasclib_cop_reg_write(cop_desc, "fat_in", fatin, arguments.adr_used,
           RASCLIB_IMMED_CMD);
       if (res != RASCLIB_SUCCESS) {
         fprintf(stderr,"reg write of fat_in failed at %d: %d\n", __LINE__, res);
         rasclib_perror("reg write of fat_in", res);
         return 1;
15     }
       rasclib_cop_commit(cop_desc, NULL);
       fatin=fatin+arguments.adr_used;
     }

20   // Receive values from RASC FPGA
     for(i=0;i<arguments.fifo/arguments.adr_used;i++){
       // do polling or not
       if(arguments.sync==TRUE){
         // Wait until all output registers are valid
25       do{
           rasclib_cop_reg_read(cop_desc, "rasc_out_vld", &rasc_out_vld,1,
               RASCLIB_IMMED_CMD);
           rasclib_cop_commit(cop_desc, NULL);
         }while(rasc_out_vld != 1);
       }
30     // Read Fat-Register-Output
       rasclib_cop_reg_read(cop_desc, "fat_out", fatout, arguments.adr_used,
           RASCLIB_IMMED_CMD);
       rasclib_cop_commit(cop_desc, NULL);
       fatout=fatout+arguments.adr_used;
     }
35 }
   end = gtod();
   printf("%f MByte/s (%f MiByte) throughput ",
                 (float)(arguments.cycles*(arguments.num_bytes/(end-start)
                     /1000000.0)),
                 (float)(arguments.cycles*(arguments.num_bytes/(end-start)
                     /1024/1024)));
40 printf("(Communication: %f sec)\n", (float)(end-start));

   // Read DEBUG Registers
   rasclib_cop_reg_read(cop_desc, "debug1", &debug[1],1, RASCLIB_IMMED_CMD);
   rasclib_cop_reg_read(cop_desc, "debug2", &debug[2],1, RASCLIB_IMMED_CMD);
45 rasclib_cop_reg_read(cop_desc, "debug3", &debug[3],1, RASCLIB_IMMED_CMD);
   rasclib_cop_reg_read(cop_desc, "debug4", &debug[4],1, RASCLIB_IMMED_CMD);
   rasclib_cop_reg_read(cop_desc, "debug5", &debug[5],1, RASCLIB_IMMED_CMD);
   rasclib_cop_reg_read(cop_desc, "debug6", &debug[6],1, RASCLIB_IMMED_CMD);
   rasclib_cop_reg_read(cop_desc, "debug7", &debug[7],1, RASCLIB_IMMED_CMD);
50 rasclib_cop_commit(cop_desc, NULL);

   for(i=1;i<8;i++) printf("debug%d = 0x%lx\t",i,debug[i]);

   DEBUG_OUT("Updated-Latency: ");
55 for(i=0;i<4;i++){
     ulatency[i*4+0] = (unsigned int)(((debug[i+1]&0xFFF000000000)>>36)-((debug[i
```

```
        +1]&0xFFF000000000000)>>48));
    ulatency[i*4+1] = (unsigned int)(((debug[i+1]&0xFFF000000)>>24)-((debug[i
        +1]&0xFFF000000000)>>36));
    ulatency[i*4+2] = (unsigned int)(((debug[i+1]&0xFFF000)>>12)-((debug[i+1]&0
        xFFF000000)>>24));
    ulatency[i*4+3] = (unsigned int)(((debug[i+1]&0xFFF)>>0)-((debug[i+1]&0
        xFFF000)>>12));
  }
  for(i=0;i<16;i++) printf("[%d]=%d; ",i,ulatency[i]);

  DEBUG_OUT("\nPolling-Latency: ");
  for(i=0;i<3;i++){
    platency[i*4+0] = (unsigned int)(((debug[i+5]&0xFFF000000000)>>36)-((debug[i
        +5]&0xFFF000000000000)>>48));
    platency[i*4+1] = (unsigned int)(((debug[i+5]&0xFFF000000)>>24)-((debug[i
        +5]&0xFFF000000000)>>36));
    platency[i*4+2] = (unsigned int)(((debug[i+5]&0xFFF000)>>12)-((debug[i+5]&0
        xFFF000000)>>24));
    platency[i*4+3] = (unsigned int)(((debug[i+5]&0xFFF)>>0)-((debug[i+5]&0
        xFFF000)>>12));
  }
  for(i=0;i<12;i++) printf("[%d]=%d; ",i,platency[i]);
  DEBUG_OUT("\n");
  // finish the algorithm ...
```

A.4 MD5-Brute-Force

Auflistung A.8: VHDL MD5-Pipeline (Ausschnitt)

```
-- md5 pipeline logic ------------------------------------------------
ramout(0) <= Dinx(0); -- 1st stage (plaintext)
pipeline: process(clk)
begin
  if clk'event and clk = '1' then -- rising edge

    -- set 2nd stage plaintext input value
    Dinx1 <= Dinx(1);
    ramout(1) <= Dinx1;

    -- stages 3 to 15
    -- ...
    -- 2nd md5 block (stages 16 to 32)
    FOR i IN 4 TO 7 LOOP
      -- stages a4 to a7
      tmp(i*4) <= md5_logic12(Dina(i)(6),Dinc(i)(2),Dind(i)(4),Dinb(i)(0),ramout(
          i*4));
      Dina(i+1)(0) <= md5_shift2(tmp(i*4),shifts(4),Dinb(i)(1),Dinhex(i*4));
      FOR s IN 0 TO 5 LOOP
        Dina(i+1)(s+1) <= Dina(i+1)(s);
      END LOOP;

      -- stages d4 to d7
      tmp(i*4+1) <= md5_logic12(Dind(i)(6),Dinb(i)(2),Dinc(i)(4),Dina(i+1)(0),
          ramout(i*4+1));
      Dind(i+1)(0) <= md5_shift2(tmp(i*4+1),shifts(5),Dina(i+1)(1),Dinhex(i*4+1))
          ;
      FOR s IN 0 TO 5 LOOP
        Dind(i+1)(s+1) <= Dind(i+1)(s);
      END LOOP;

      -- stages c4 to c7
      tmp(i*4+2) <= md5_logic12(Dinc(i)(6),Dina(i+1)(2),Dinb(i)(4),Dind(i+1)(0),
          ramout(i*4+2));
      Dinc(i+1)(0) <= md5_shift2(tmp(i*4+2),shifts(6),Dind(i+1)(1),Dinhex(i*4+2))
          ;
      FOR s IN 0 TO 5 LOOP
        Dinc(i+1)(s+1) <= Dinc(i+1)(s);
      END LOOP;

      -- stages b4 to b7
      tmp(i*4+3) <= md5_logic12(Dinb(i)(6),Dind(i+1)(2),Dina(i+1)(4),Dinc(i+1)(0)
          ,ramout(i*4+3));
      Dinb(i+1)(0) <= md5_shift2(tmp(i*4+3),shifts(7),Dinc(i+1)(1),Dinhex(i*4+3))
          ;
      FOR s IN 0 TO 5 LOOP
        Dinb(i+1)(s+1) <= Dinb(i+1)(s);
      END LOOP;
    END LOOP;

    -- stages 33 to 64
    -- ...
  end if;
end process;
```

Auflistung A.9: MD5 Hash-Vergleich in VHDL

```
-- hash comparison
proc_cmp: process(clk)
begin
  if clk'event and clk = '1' then -- rising edge
    eqhash <= (others => '0');
    -- 1st md5 checksum part available
    if pipe_vld = '1' AND douthash(0) = Din_hash(127 downto 96) then
      eqhash(0) <= '1';
    end if;
    rdy_part(0) <= eqhash(0);
    -- 2 clocks later
    if rdy_part(0) = '1' AND douthash(3) = Din_hash(31 downto 0) then
      eqhash(3) <= '1';
    end if;
    rdy_part(1) <= eqhash(3);
    -- 4 clocks later
    if rdy_part(1) = '1' AND douthash(2) = Din_hash(63 downto 32) then
      eqhash(2) <= '1';
    end if;
    rdy_part(2) <= eqhash(2);
    -- 6 clocks later
    if rdy_part(2) = '1' AND douthash(1) = Din_hash(95 downto 64) then
      eqhash(1) <= '1';
    end if;
  end if;
end process;

-- signal if pipe has found the hash
proc_equal: process(clk)
begin
  if clk'event and clk = '1' then -- rising edge
    if eqhash(1) = '1' then
      Vout <= '1';
    else
      Vout <= '0';
    end if;
  end if;
end process;
```

Auflistung A.10: MD5 Klartextrückgewinnung in VHDL

```
-- the range of ASCII characters to be processed
constant firstchar : positive
  := conv_integer(conv_unsigned(character'POS(' '),8));
constant lastchar : positive
  := conv_integer(conv_unsigned(character'POS('~'),8));
-- std_logic_vectors for first and last ASCII-character
constant vfirstchar : std_logic_vector(7 downto 0)
  := CONV_STD_LOGIC_VECTOR(firstchar,8);
constant vlastchar : std_logic_vector(7 downto 0)
  := CONV_STD_LOGIC_VECTOR(lastchar,8);
-- number of ASCII-characters to be processed
constant characters : positive := lastchar-firstchar+1;
-- the range of ASCII-characters for the first plaintext-character has to be a
    multiple of PIPES
constant pipes_norm : positive := getNormPipes(firstchar,lastchar,PIPES);
constant pipes_ov : natural := pipes-pipes_norm;
constant realchars : positive := characters + pipes_ov;
constant vrlastchar : std_logic_vector(7 downto 0)
  := CONV_STD_LOGIC_VECTOR(lastchar + pipes_ov,8);
-- first char has to be reduced by:
constant chardiff0: std_logic_vector(7 downto 0)
```

```
  := CONV_STD_LOGIC_VECTOR((131 * PIPES mod realchars),8);
-- second char has to be reduced by:
constant chardiff1: std_logic_vector(7 downto 0)
  := CONV_STD_LOGIC_VECTOR(131 * PIPES / realchars,8);

-- does only work if more than 33 different characters are used
proc_cmp:process(clk)
begin
  if clk'event and clk = '1' then -- rising edge

    -- check which pipe has found the hash and create plaintext
    vld_cmp <= '0';
    FOR p IN 0 TO PIPES-1 LOOP
      if vout_pipe(p) = '1' then
        vpipe <= p;
        vld_cmp <= '1';
        -- save all characters
        -- first two characters can already be adjusted (pipeline depth)
        char(0) <= charctrs((CHARS)*8-1 downto (CHARS-1)*8)
                   - CONV_STD_LOGIC_VECTOR(PIPES-1,8) - chardiff0;
        char(1) <= charctrs((CHARS-1)*8-1 downto (CHARS-2)*8) - chardiff1;
        FOR c IN 2 TO CHARS-1 LOOP
          char(c) <= charctrs((CHARS-c)*8-1 downto (CHARS-c-1)*8);
        END LOOP;
      end if;
    END LOOP;

  end if; --clock
end process;

proc_plainout:process(clk)
begin
  if clk'event and clk = '1' then -- rising edge
    -- set the plaintext of unused characters
    dout_plain(447 downto CHARS*8) <= (others => '0');
    if rst = '1' then
      cvld <= (others => '0');
    else

      if vld_cmp = '1' then
        -- process char0
        if char(0) < vfirstchar then
          dout_plain((CHARS)*8-1 downto (CHARS-1)*8) <=
            vrlastchar - (vfirstchar - char(0))
            + CONV_STD_LOGIC_VECTOR(vpipe,8);
          chartmp(1) <= char(1) - 1;
        else
          dout_plain((CHARS)*8-1 downto (CHARS-1)*8) <=
            char(0) + CONV_STD_LOGIC_VECTOR(vpipe-1,8);
          chartmp(1) <= char(1);
        end if;
        -- char0 plaintext is valid
        cvld(0) <= '1';
      end if;

      --process char1
      if cvld(0) = '1' then
        if chartmp(1) < vfirstchar then
          dout_plain((CHARS-1)*8-1 downto (CHARS-2)*8) <= characters + chartmp(1)
              ;
          chartmp(2) <= char(2) - 1;
        else
          dout_plain((CHARS-1)*8-1 downto (CHARS-2)*8) <= chartmp(1);
```

```
          chartmp(2) <= char(2);
        end if;
        cvld(1) <= '1';
      end if;

      -- set chars 2 to penultimate
      FOR c IN 2 TO CHARS-2 LOOP
        if cvld(c-1) = '1' then
          if chartmp(c) < vfirstchar then
            dout_plain((CHARS-c)*8-1 downto (CHARS-c-1)*8) <= vlastchar;
            chartmp(c+1) <= char(c+1)-1;
          else
            dout_plain((CHARS-c)*8-1 downto (CHARS-c-1)*8) <= chartmp(c);
            chartmp(c+1) <= char(c+1);
          end if;
          cvld(c) <= '1';
        end if;
      END LOOP;

      -- set last char
      if cvld(CHARS-2) = '1' then
        if chartmp(CHARS-1) < vfirstchar then
          dout_plain(7 downto 0) <= vlastchar;
        else
          dout_plain(7 downto 0) <= chartmp(CHARS-1);
        end if;
        cvld(CHARS-1) <= '1';
      end if;
    end if; --rst
  end if;
end process;
-- if last char is processed, set output valid
vld <= cvld(CHARS-1);
```

Auflistung A.11: Datenübertragung MD5-Brute-Force Hostprogramm

```
// stelle zu suchenden Hash und Startklartext bereit ...

// starte den FPGA
if((res = startAlgorithm(&cop_desc)) < RASCLIB_SUCCESS){
  fprintf(stderr,"Could not start Algorithm. Error Code: %d\n",res);
  rasclib_perror("Algorithm could not be started.", res);
  return 1;
}

start = gtod(); // speichere Startzeit

// sende zu suchenden Hash
res = rasclib_cop_reg_write(cop_desc, "md5hash", din_hash, SIZE_HASH,
    RASCLIB_IMMED_CMD);
if (res != RASCLIB_SUCCESS) {
  fprintf(stderr,"reg write of md5hash failed at %d: %d\n", __LINE__, res);
  rasclib_perror("reg write of md5hash", res);
  return 1;
}
// sende Startklartext
res = rasclib_cop_reg_write(cop_desc, "md5init", din_init, SIZE_INIT,
    RASCLIB_IMMED_CMD);
if (res != RASCLIB_SUCCESS) {
  fprintf(stderr,"reg write of md5init failed at %d: %d\n", __LINE__, res);
  rasclib_perror("reg write of md5init", res);
  return 1;
}

```

```
// starte die Suche
start_md5 = 1;
res = rasclib_cop_reg_write(cop_desc, "start", &start_md5, 1, RASCLIB_IMMED_CMD);
if (res != RASCLIB_SUCCESS) {
  fprintf(stderr,"reg write of start failed at %d: %d\n", __LINE__, res);
  rasclib_perror("reg write of start", res);
  return 1;
}
rasclib_cop_commit(cop_desc, NULL);

// Warte, bis der Hash gefunden oder alle Moeglichkeiten durchprobiert wurden
do {
  rasclib_cop_reg_read(cop_desc, "rasc_out_vld", &rasc_out_vld, 1,
      RASCLIB_IMMED_CMD);
  rasclib_cop_commit(cop_desc, NULL);
} while (rasc_out_vld != 1);

// lese den Klartext
rasclib_cop_reg_read(cop_desc, "plain448", dout_plain, SIZE_PLAIN,
    RASCLIB_IMMED_CMD);
rasclib_cop_commit(cop_desc, NULL);

end = gtod(); // speichere Endzeit

// gebe Klartext aus
DEBUG_OUT("Plaintext as 64bit hexadecimal values: \n");
for(i=0;i<SIZE_PLAIN;i++){
  printf("0x%lx - ",dout_plain[i]);
  long2String((unsigned long long *)&dout_plain[i]);
  DEBUG_OUT("\n");
}

// gebe den Zeitbedarf und die Hashes pro Sekunde aus
printf("Die Programmlaufzeit betrug %lf Sekunden\n", (double)(end-start));
// no. of hashes created:
// 1st char: 96 characters (multiple of implemented pipelines)
// 2nd to 5th char: ASCII 32 to 126 (95 characters)
printf("%.3lfM Hashes/sec\n",96*pow(95,4)/1000000.0/(end-start));

// beende den Algorithmus
finish_alg = 1;
res = rasclib_cop_reg_write(cop_desc, "finish_alg", &finish_alg, 1,
    RASCLIB_IMMED_CMD);
if (res != RASCLIB_SUCCESS) {
  fprintf(stderr,"reg write of finish_alg failed at %d: %d\n",__LINE__, res);
  rasclib_perror("reg write of finish_alg", res);
  return 1;
}

// gebe den FPGA frei
if((res = freeFPGA(&cop_desc, alg_name)) < RASCLIB_SUCCESS){
  fprintf(stderr,"Could not release FPGA. Error Code: %d\n",res);
  rasclib_perror("FPGA could not be released.", res);
  return 1;
}
```

A.5 Damenproblem

Auflistung A.12: Erster Versuch einer Mitrion-C-Implementierung des Damenproblems

```
Mitrion-C 1.5;
// options: -cpp

#define N 26
#define L   6
#define L_1 5
#define NL  20

#define SLICES 1
#define PREPLACEMENTS 1
#define SEQ_PRE 1 //PREPLACEMENTS/SLICES

#define ExtRAM  mem bits:128[2048]

/**Calculates all completions of valid Queens placements on a NxN board
 * under the pre-placement provided in cs as:
 *   |...|col(0)|col(1)|...|col(L-1)|        col(i): 5 bits
 * This encoding is valid up to L = 6 and N=32 when col(0) is placed only
 * up to the middle row - thus requiring, at most, 4 bits.
 * @param   cs  Pre-Placement
 * @return  Number of valid Board Completions. (46 Bit should be enough)
 */
uint:46 solveQueens(uint:32 cs){

  mem_bh0 = memcreate(mem uint:32[N] mem_bh_last); // blocking horizontal
  mem_bu0 = memcreate(mem uint:32[N] mem_bu_last); // blocking upwards
  mem_bd0 = memcreate(mem uint:32[N] mem_bd_last); // blocking downwards
  mem_sl0 = memcreate(mem uint:32[N] mem_sl_last); // free slots yet to try

  // Initialize Blocking
  uint:32  bh0 = 0;
  uint:32  bu0 = 0;
  uint:32  bd0 = 0;
  uint:32 zero = 0;
  // get blocking from pre-placement
  (uint:32,uint:32,uint:32)(bh1, bu1, bd1) = for(i in < 0 .. L_1 >) {
    uint:32 q = (uint:32)1 << (0x1F & (cs >> (uint:32)((L-i-1)*5)));
    bh0 = bh0 | q;
    bu0  = (uint:32)(bu0|q) << 1;
    bd0  = (bd0|q) >> 1;
  }(bh0,bu0,bd0);

  mem_bh1 = memwrite(mem_bh0,L,bh1);
  mem_bu1 = memwrite(mem_bu0,L,bu1);
  mem_bd1 = memwrite(mem_bd0,L,bd1);

  uint:32 sl = ~(((uint:32)(~((uint:32)0)) << N)| bh1 | bu1 | bd1);
  mem_sl1 = memwrite(mem_sl0,L,sl);
  tuple{mem uint:32[N], mem uint:32[N], mem uint:32[N], mem uint:32[N]} mem_loop
    = tup(mem_bh1, mem_bu1, mem_bd1, mem_sl1);

  // Explore Solutions
  uint:46 cnt0 = 0;
  int:32 i = L;
  (cnt,mem_al) = while(i >= L) {
    (mem_bht, mem_but, mem_bdt, mem_slt) = untup(mem_loop);
    (slot,mem_slt1) = memread(mem_slt,i);
    mem_loop1 = tup(mem_bht, mem_but, mem_bdt, mem_slt1);
    (i,cnt0,mem_loop) = if(slot == 0) {
```

```
      uint:32 inn = i-1;
    } (inn, cnt0, mem_loop1)
    else {
      sloti = slot & (uint:32)-slot;
      (itmp,cnttmp, mem_looptmp) = if(i < (N-1)) {
        p = i;
        int:32 ip1 = i +1;
        (slp,mem_slt2)= memread(mem_slt1,i);
        slpp = slp ^ sloti;
        mem_slt3 = memwrite(mem_slt2, i, slpp);

        (bhp,mem_bht1) = memread(mem_bht,i);
        (bup,mem_but1) = memread(mem_but,i);
        (bdp,mem_bdt1) = memread(mem_bdt,i);

        bhps = bhp | sloti;
        bups = (uint:32)(bup | sloti) << 1;
        bdps = (bdp | sloti) >> 1;

        mem_bht2 = memwrite(mem_bht1,ip1,bhps);
        mem_but2 = memwrite(mem_but1,ip1,bups);
        mem_bdt2 = memwrite(mem_bdt1,ip1,bdps);

        sslip1 = ~(((uint:32)(~((uint:32)0)) << N)|bhps|bups|bdps);

        mem_slt4 = memwrite(mem_slt3,ip1,sslip1);
        mem_loop2 = tup(mem_bht2, mem_but2, mem_bdt2, mem_slt4);
      }(ip1,cnt0,mem_loop2)
      else{
        // Count Solution and proceed with preceding Column
        uint:46 cnt0n = cnt0 + 1;
        watch cnt0n;
        int:32 inn=N-2;
      }(inn,cnt0n,mem_loop1);
    }(itmp,cnttmp,mem_looptmp);
  }(cnt0,mem_loop);
  (mem_bh_last, mem_bu_last, mem_bd_last, mem_sl_last) = untup(mem_al);
}(cnt);

(ExtRAM, ExtRAM) main(ExtRAM sram0, ExtRAM sram1){
  // read pre-placements from SRAM0
  (sram0pre,listpre) = foreach(idx in <0 .. PREPLACEMENTS-1>){
    (pre128,sram0_pre) = memread(sram0,idx);
    bits:32 pre32 = (bits:32)pre128;
  }(sram0_pre,pre32);
  // define number of slices and sequential execution
  prepls2 = reshape(listpre, <SLICES><SEQ_PRE>);
  prepls3 = reformat(prepls2,[SLICES]<SEQ_PRE>);
  sram1all = foreach(pres_elem in prepls3 by idx){
    sram1slices = foreach(pre_elem in pres_elem by idx_pre){
      uint:46 psol = solveQueens((uint:32)pre_elem);
      psol128 = (uint:128)psol;
      sram1ps = memwrite(sram1,idx*SEQ_PRE+idx_pre,psol128);
      watch psol;
    }sram1ps;
  }sram1slices;
}(sram0pre,sram1all);
```

Auflistung A.13: Bandbreitengraph Damenproblem Mitrion-C (1)

```
------------------------- Summary: Bandwidth Graph -------------------------
ROOT
|  l: 6.0 ch: 6 rch : 6  BW limited. body ch: 6 rch: 6
   |--1
   | iterations: 1
   |   Env/main/foreach<1>
   |  l: 1.0 ch: 1 rch : 6  BW limited. body ch: 1 rch: 6
   |  |--1
   |  | iterations: 1
   |  |  l: 1.0 ch: 1 rch : 6  BW limited. body ch: 1 rch: 6
   |--2
   | iterations: 1
   |   Env/main/foreach[1]/foreach<1>
   |  l: 6.0 ch: 6 rch : 6  BW limited. body ch: 6 rch: 6
      |--1
      | iterations: 6
      |   ...nv/main/foreach[1]/foreach<1>/solveQ26/for<6>
      |  l: 60.0 ch: 6 rch : 6  Latency limited dataloop. body latency: 10.0
      |--2
      | iterations: 1
      |   Env/main/foreach[1]/foreach<1>/solveQ26/while
      |  l: 15.0 ch: 1 rch : 6  Latency limited dataloop. body latency: 15.0
------------------------ End Summary: Bandwidth Graph ------------------------
```

Auflistung A.14: Optimierte Mitrion-C-Implementierung des Damenproblems

```
Mitrion-C 1.5;
// options: -cpp

#define N 26
#define L   6
#define NNN 78  // N*3

#define SLICES 2
#define PREPLACEMENTS 16
#define PERSLICE 8

/**Calculates all completions of valid Queens placements on a NxN board
 * under the pre-placement provided in cs as:
 *   |...|col(0)|col(1)|...|col(L-1)|        col(i): 5 bits
 * This encoding is valid up to L = 6 and N=32 when col(0) is placed only
 * up to the middle row - thus requiring, at most, 4 bits.
 * @param   cs  Pre-Placement
 * @return  Number of valid Board Completions. (46 Bit should be enough)
 */
uint:46 solveQueens(uint:32 cs){
  // create internal RAM (Block-RAM)
    // blocking (horizontal, diagonal up, diagonal down)
    mem_b0 = memcreate(mem bits:NNN[N] mem_b_last);
    mem_sl0 = memcreate(mem bits:N[N] mem_sl_last); // free slots yet to try

  // Initialize Blocking
  bits:N  bh0 = 0;
  bits:N  bu0 = 0;
  bits:N  bd0 = 0;

  (bits:N,bits:N,bits:N)(bh1, bu1, bd1) = for(i in < 0 .. L-1 >) {
    uint:N q = (uint:N)1 << (0x1F & (cs >> (uint:32)((L-i-1)*5)));
    bh0 = bh0 | q;
    bu0  = (bits:N)(bu0|q) << 1;
    bd0  = (bd0|q) >> 1;
  }(bh0,bu0,bd0);
  mem_b1 = memwrite(mem_b0,L,[bh1,bu1,bd1]);

  bits:N sl = ~(bh1|bu1|bd1);
  mem_sl1 = memwrite(mem_sl0,L,sl);

  tuple{mem bits:NNN[N], mem bits:N[N]} mem_loop = tup(mem_b1, mem_sl1);

  // Explore Solutions
  uint:46 cnt0 = 0;
  int:6 i = L;
  (cnt,mem_al) = while(i >= L) {

    (mem_bt, mem_slt) = untup(mem_loop);
    // 3 cycles delay
    (slot,mem_slt1) = memread(mem_slt,i); //watch slot;
    // 1 cycle delay
    (bp,mem_bt1) = memread(mem_bt,i);
    bits:N[3] bparray = bp;
    (bhp, bup, bdp) = (bparray[0],bparray[1],bparray[2]);

    // if free slot has been found and not last column
    (mem_loop) = if((slot != (bits:N)0) && (i < (N-1))){
      // prepare slots and blocking (2 cycles)
      nslot = slot & (uint:N)-((uint:N)slot); // next slot to try
        bits:N cslots = slot ^ nslot; //watch cslots; // set current column slots
        // set next column blocking and slots
```

```
        bits:N bh_nc = bhp | nslot;
        bits:N bu_nc = (bits:N)(bup | nslot) << 1;
        bits:N bd_nc = (bdp | nslot) >> 1;
        slots_nc = ~(bh_nc|bu_nc|bd_nc); //watch slots_nc;

      // write blocking and slot (2 cycles)
      mem_bt2 = memwrite(mem_bt1,(int:6)(i+1),[bh_nc, bu_nc, bd_nc]);
      mem_slt2 = memwrite(mem_slt1, i, cslots);
      mem_slt3 = memwrite(mem_slt1,(int:6)(i+1),slots_nc);
      mem_loop2 = tup(mem_bt2, mem_slt3);
    }(mem_loop2)
    else (mem_loop);

    // set next i (loop dependend variable) and solution counter
    (i,cnt0) = if(slot == (bits:N)0){
      int:6 im1 = i-1;
    }(im1, cnt0)
    else{
      (itmp,cnttmp) = if(i < (N-1)) { // is last column?
        int:6 ip1 = i + 1;
      }(ip1,cnt0)
      else{
        // Count Solution and proceed with preceding Column
        uint:46 cnt0n = cnt0 + 1; //watch cnt0n;
        int:6 inn=N-2;
      }(inn,cnt0n);
    }(itmp,cnttmp);

  }(cnt0,mem_loop);
  (mem_b_last, mem_sl_last) = untup(mem_al);

}(cnt);

#define ExtRAM  mem bits:128[2048]
(ExtRAM, ExtRAM) main(ExtRAM sram0, ExtRAM sram1){
  (sram0all,sram1all) = foreach(idx_slc in [0 .. SLICES - 1]){
    (sram0slices, sram1slices) = foreach(idx_pre in <0 .. PERSLICE - 1>) {
      idx = idx_slc*PERSLICE+idx_pre;
      (pre128, sram0ps) = memread(sram0,idx);
      bits:32 pre32 = (bits:32)pre128;
      uint:46 psol = solveQueens((uint:32)pre32);
      psol128 = (uint:128)psol; watch psol;
      sram1ps = memwrite(sram1,idx,psol128);
    }(sram0ps,sram1ps);
  }(sram0slices,sram1slices);
}(sram0all,sram1all);
```

Auflistung A.15: Damenproblem: Native C-Funktionen

```
#include <jni.h>
#include "stdlib.h"
#include "string.h"
#include "rasc/rasclib.h"

// for queens we are using bufferd IO, because communication is not critical
#define IO_TYPE RASCLIB_BUFFERED_IO
#define ERROR_TIMEOUT (-51)

int freeFPGA(int *cop_desc, char const *const algID);

/* return value: COP descriptor (SUCCESS) or error code (FAILURE) */
JNIEXPORT jint JNICALL Java_jrasc_JRasc_reserveFPGA(JNIEnv *jenv,
        jclass const clsJRasc, jstring jalgname) {
  jint res;
  jboolean iscopy;
  char const *const astring = (*jenv)->GetStringUTFChars(jenv,jalgname,&iscopy);
  char const *const ustring = getenv("USER");

  /********************* RASC FPGA Initialization ************************/
  // reserve one FPGA (ustring or resrv_name can be any string)
  if((res = rasclib_resource_reserve(1, ustring)) == RASCLIB_SUCCESS){
    if((res = rasclib_resource_configure(astring, 1, ustring)) == RASCLIB_SUCCESS
        ){
      // get descriptor for reserved FPGA in res
      if((res = rasclib_cop_open(astring, IO_TYPE)) == RASCLIB_SUCCESS) {
        unsigned long finish_alg = 0;
        if((res = rasclib_cop_reg_write(res, "finish_alg", &finish_alg, 1,
            RASCLIB_IMMED_CMD)) == RASCLIB_SUCCESS) {
          rasclib_cop_commit(res, NULL);
          rasclib_cop_go(res);
          unsigned long reset;
          do{
            rasclib_cop_reg_read(res, "reset", &reset, 1, RASCLIB_IMMED_CMD);
            rasclib_cop_commit(res, NULL);
          }while (reset);
          // Wait 5s until reset is done
          jclass const clsThread = (*jenv)->FindClass(jenv, "java/lang/Thread");
          jmethodID const mthSleep = (*jenv)->GetStaticMethodID(jenv, clsThread,
              "sleep", "(J)V");
          (*jenv)->CallStaticVoidMethod(jenv,clsThread,mthSleep,(jlong)5000L);
          (*jenv)->ExceptionClear(jenv);
        }
      }
    }
  }

  // release not any more needed objects
  (*jenv)->ReleaseStringUTFChars(jenv, jalgname, astring);
  return res;
}

// receives values from fpga and returns them as byte array
JNIEXPORT jint JNICALL Java_jrasc_JRasc_recvByteArray(JNIEnv *const jenv,
        jclass const clsJRasc, jint jcop_desc, jbyteArray const jbarray) {

  jsize jb_length = (*jenv)->GetArrayLength(jenv, jbarray);
   // Algorithm defined register array length
  const int ADR_SIZE = jb_length/ADR_BYTES;
  unsigned long rasc_out_vld;
  unsigned long fat_out512[ADR_SIZE];
  unsigned char byteout[jb_length];
```

```
  int res;
  int j,i,timeout;

  // Reduced polling
  while(true) {
    // Check Availability of Data
    res = rasclib_cop_reg_read(jcop_desc, "rasc_out_vld", &rasc_out_vld, 1,
        RASCLIB_IMMED_CMD);
    if (res != RASCLIB_SUCCESS) return res;

    rasclib_cop_commit(jcop_desc, NULL);
    if(rasc_out_vld == 1)  break;

    // Wait 30s before next Re-Try
    jclass    const  clsThread = (*jenv)->FindClass(jenv, "java/lang/Thread");
    jmethodID const  mthSleep  = (*jenv)->GetStaticMethodID(jenv, clsThread, "
        sleep", "(J)V");
    (*jenv)->CallStaticVoidMethod(jenv, clsThread, mthSleep, (jlong)10000L);
    (*jenv)->ExceptionClear(jenv);
  }

  /***** Read Fat-Register-Output (4 64Bit Registers) *****/
  res = rasclib_cop_reg_read(jcop_desc, "out256", fat_out512, ADR_SIZE,
      RASCLIB_IMMED_CMD);
  if (res != RASCLIB_SUCCESS) {
    return res;
  }
  rasclib_cop_commit(jcop_desc, NULL);

  for (j = 0; j < ADR_SIZE; j++) {
    int pos = j * 8;
    byteout[pos + 7] = fat_out512[j] & 0xFF;
    byteout[pos + 6] = (fat_out512[j] & 0xFF00) >> 8;
    byteout[pos + 5] = (fat_out512[j] & 0xFF0000) >> 16;
    byteout[pos + 4] = (fat_out512[j] & 0xFF000000) >> 24;
    byteout[pos + 3] = (fat_out512[j] & 0xFF00000000) >> 32;
    byteout[pos + 2] = (fat_out512[j] & 0xFF0000000000) >> 40;
    byteout[pos + 1] = (fat_out512[j] & 0xFF000000000000) >> 48;
    byteout[pos] = (fat_out512[j] & 0xFF00000000000000) >> 56;
  }

  // 3rd and 4th arguments are start element and array length
  (*jenv)->SetByteArrayRegion(jenv, jbarray, 0, jb_length, (jbyte *) byteout);
  return RASCLIB_SUCCESS;
}

// send byte array as 64Bit values to FPGA
JNIEXPORT jint JNICALL Java_jrasc_JRasc_sendByteArray(JNIEnv *jenv,
        jobject jobj, jint jcop_desc, jbyteArray jbarray) {

  jsize jb_length = (*jenv)->GetArrayLength(jenv, jbarray);
  int ADR_SIZE = jb_length/ADR_BYTES;
  jboolean isCopy;
  signed char *carray = (*jenv)->GetByteArrayElements(jenv, jbarray, &isCopy);
  int j;
  int res;
  unsigned long fat_in128[ADR_SIZE];

  for (j = 0; j < ADR_SIZE; j++) {
    // does only work for unsigned long = 64Bit (e.g. IA64)
    fat_in128[j] = (((unsigned long) ((*carray)&0xff) << 56) |
                ((unsigned long) ((*(carray + 1))&0xff) << 48) |
                ((unsigned long) ((*(carray + 2))&0xff) << 40) |
```

```
                ((unsigned long) ((*(carray + 3))&0xff) << 32) |
                ((unsigned int) ((*(carray + 4))&0xff) << 24) |
                ((unsigned int) ((*(carray + 5))&0xff) << 16) |
                ((unsigned int) ((*(carray + 6))&0xff) << 8) |
                ((*(carray + 7))&0xff));
    carray+=8;
  }

  // Send RASC FPGA Input
  res = rasclib_cop_reg_write(jcop_desc, "in64", fat_in128, ADR_SIZE,
      RASCLIB_IMMED_CMD);
  if (res != RASCLIB_SUCCESS) {
    return res;
  }
  rasclib_cop_commit(jcop_desc, NULL);
  return RASCLIB_SUCCESS;
}

JNIEXPORT jint JNICALL Java_jrasc_JRasc_freeFPGA(JNIEnv *jenv,
        jobject jobj, jint jcop_desc, jstring jalgname) {
  jboolean iscopy;
  char const *const astring = (*jenv)->GetStringUTFChars(jenv,jalgname,&iscopy);
  return freeFPGA(&jcop_desc, astring);
}

int freeFPGA(int *cop_desc, char const *const algID){
  int res;
  char const *const ustring = getenv("USER");
  unsigned long debug1;
  unsigned long finish_alg = 1;
  if ((res = rasclib_cop_reg_write(*cop_desc, "finish_alg", &finish_alg, 1,
      RASCLIB_IMMED_CMD)) == RASCLIB_SUCCESS) {
    rasclib_cop_commit(*cop_desc, NULL);
    rasclib_cop_wait(*cop_desc);
    rasclib_cop_close(*cop_desc);
    rasclib_resource_return(algID, 1);
    rasclib_resource_release(1, ustring);
  }
  return res;
}
```

Auflistung A.16: Damenproblem: Java Interface

```
package jrasc;

import java.io.InputStream;
import java.io.OutputStream;
import java.util.Date;

/**Provides access to RASC FPGAs. */
public class JRasc{

  // if library cannot be found -> UnsatisfiedLinkError
  static {
      System.loadLibrary("rascjni");
  }

  /* The paket size in bytes to be send */
  private final int inputBytes;
  /* The paket size in bytes to be received */
  private final int outputBytes;

  /* Handle to the represented FPGA configuration. */
  private final int hdl;
```

```
  /* The identifier of the algorithm to be loaded to the FPGA */
  private final String alg_name;
  private OutputStream out;
  private InputStream in;

  // performance evaluation
  private long starttime;
  private long subboards;

  private JRasc(int hdl, String alg, int inputBytes, int outputBytes) {
    this.hdl = hdl;
    this.alg_name = alg;
    this.inputBytes = inputBytes;
    this.outputBytes = outputBytes;
    this.starttime = new Date().getTime();
    this.subboards = 0;
  }

  private static native int reserveFPGA(String alg);
  private static native int freeFPGA(int cop_desc, String alg);
  private static native int sendByteArray(int cop_desc, byte[] input);
  private static native int recvByteArray(int cop_desc, byte[] output);

  /**
   * Reserve an FPGA and return or representing JRasc instance.
   * @param alg the algorithm to be loaded
   * @param inputBytes the paket size in bytes to be send
   * @param outputBytes the paket size in bytes to be received
   * @throws JRascException on failure.
   */
  public static JRasc reserve(String alg, int inputBytes, int outputBytes)
          throws JRascException {

    final int fd = reserveFPGA(alg);
    if (fd < 0) {
      throw new JRascException(fd);
    }
    return new JRasc(fd,alg,inputBytes,outputBytes);
  }

  /** Releases the RASC device
   *  @return 0 if successfully or error code
   */
  public int free(){
    return freeFPGA(this.hdl,this.alg_name);
  }

  /** Returns an OutputStream to send data to RASC
   * @return an OutputStream
   */
  public synchronized OutputStream getOutputStream() {
    final OutputStream res = out;
    if (res != null) {
      return res;
    }
    return out = new OutputStream() {

      private byte[] buf = new byte[JRasc.this.inputBytes];
      private int idx;

      public synchronized void write(int b) throws JRascException {
        buf[idx++] = (byte)b;
        if (idx == JRasc.this.inputBytes) {
```

```
          int res = sendByteArray(hdl, buf);
          if(res<0){
            throw new JRascException(res);
          }
          idx = 0;
        }
      }
    };
  }

  /** Returns an InputStream to receive data from RASC
   * @return an InputStream
   */
  public synchronized InputStream getInputStream() {
    final InputStream res = in;
    if (res != null) {
      return res;
    }
    return in = new InputStream() {

      private byte[] buf = new byte[JRasc.this.outputBytes];
      private int idx = buf.length;

      public synchronized int read() throws JRascException {
        if (idx >= JRasc.this.outputBytes) {
          int res = recvByteArray(hdl, buf);
          if(res<0){
              throw new JRascException(res);
          }
          idx = 0;
          // performance evaluation
          JRasc.this.subboards += outputBytes/16;
          long elapsed = new Date().getTime()-starttime;
          System.err.println((float)JRasc.this.subboards/(elapsed/3600000.0)
            + "subboards/hour");
        }
        return (buf[idx++]&0xff);
      }
    };
  }
}
```

Erklärungen zum Urheberrecht

Silicon Graphics, SGI und Altix sind eingetragene Marken und NUMAlink, ProPack und SGI RASC sind Marken von Silicon Graphics, Inc., in den U.S. und/oder anderen Ländern weltweit.

Mitrion-C, Mitrion Virtual Processor und Mitrion Software Development Kit sind Marken und Mitrion ist eine eingetragene Marke von Mitrionics AB.

Impulse C ist eine Marke von Impulse Accelerated Technologies, Inc.

Handel-C ist eine Marke von Agility Design Solutions Inc.

Intel, Itanium und Core2Duo sind Marken der Intel Corporation oder ihrer Tochtergesellschaften in den USA oder anderen Ländern. Dies gilt auch, wenn es nicht explizit im Text gekennzeichnet ist.

NVIDIA und CUDA sind Marken der NVIDIA Corporation in den USA oder anderen Ländern. Dies gilt auch, wenn es nicht explizit im Text gekennzeichnet ist.

Xilinx, Virtex und Xilinx - ISE sind Marken von Xilinx in den USA oder anderen Ländern. Dies gilt auch, wenn es nicht explizit im Text gekennzeichnet ist.

Synplify Pro ist eine Marke von Synplicity. Dies gilt auch, wenn es nicht explizit im Text gekennzeichnet ist.

AMD, Opteron, Phenom, AthlonX2, Brook und Radeon sind Marken von Advanced Micro Devices. Dies gilt auch, wenn es nicht explizit im Text gekennzeichnet ist.

Linux ist eine eingetragene Marke von Linus Torvalds. Dies gilt auch, wenn es nicht explizit im Text gekennzeichnet ist.

Andere Marken oder Produktnamen sind Eigentum der jeweiligen Inhaber.